Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

Uma abordagem para verificação de missões multi-robôs em alto nível no UPPAAL

Danilo B. Galvão

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof.a Dr.a Genaina Nunes Rodrigues

Brasília
2023

Ficha Catalográfica de Teses e Dissertações

Está página existe apenas para indicar onde a ficha catalográfica gerada para dissertações de mestrado e teses de doutorado defendidas na UnB. A Biblioteca Central é responsável pela ficha, mais informações nos sítios:

http://www.bce.unb.br
http://www.bce.unb.br/elaboracao-de-fichas-catalograficas-de-teses-e-dissertacoes

Esta página não deve ser inclusa na versão final do texto.

Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

Uma abordagem para verificação de missões multi-robôs em alto nível no UPPAAL

Danilo B. Galvão

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof.a Dr.a Genaina Nunes Rodrigues (Orientador)
CIC/UnB

Prof. Dr. Rodrigo Bonifácio   Prof. Dr. Radu Calinescu
CIC/UnB   University of York

Prof. Dr. Ricardo Jacobi
Coordenador do Programa de Pós-graduação em Informática

Brasília, 19 de janeiro de 2023

Dedicatória

Dedico esse a minha família: Maria Madalena, João Vitor e José Pereira. O apoio emocional de alguns de vocês me deu energias quando eu não conseguia me levantar sozinho. Dedico esse trabalho também à minha namorada e parceira de mestrado Helena Schubert, que me consolou e me deu forças nos momentos mais difíceis.

Agradecimentos

A pandemia de 2020 significou um aumento de dificuldades para muitos estudantes, agravado por um governo dedicado a destruir muitas instituições vitais para o desenvolvimento social. A educação foi uma de suas maiores vítimas por constantes cortes em verbas do orçamento da educação. Como estudante, me senti diversas vezes desmotivado e ameaçado durante o processo. Meu primeiro agradecimento é direcionado a todos os acadêmicos que me mostraram que a verdadeira ciência sempre foi resistência em um país onde nossos representantes estão mais preocupados em manutenção do poder e sobrevivência própria. Alguns nomes muito importantes são a minha orientadora, um exemplo de pesquisadora, Genaina Nunes Rodrigues. Além de todos os outros professores que me ensinaram algo durante a jornada como Raian Ali, Bozena Wozna-Szczesniak e vários outros que não estão aqui nominalmente mas sempre farão parte das pessoas que fizeram a diferença. Um agradecimento especial a alguns colegas de pesquisa como o Eric, Artur e Gabriel que me ajudaram imensamente durante o desenvolvimento do trabalho. Em segundo, agradeço a bolsa oferecida pela CAPES, que me permitiu fazer a compra do computador onde eu fiz a maioria desse trabalho.

Abstract

Title: An Approach for High-Level Multi Robot Mission Verification in UPPAAL

The need to leverage means to specify robotic missions from a high abstraction level has gained momentum due to the popularity growth of robotic applications. As such, it is paramount to provide means to guarantee that not only the robotic mission is correctly specified, but that it also guarantees degrees of safety given the growing complexity of tasks assigned to Multi-Robot System (MRS). Therefore, robot missions now need to be specified and formally verified for both robots and other agents involved in the robotic mission operation. However, many mission specifications lack a streamlined verification process that ensures that all mission properties are thoroughly verified through model checking. This work proposes a model checking process for mission specification and decomposition of MRS in UPPAAL model checker. In particular, we present an automated generation process containing hierarchical domain definition properties transformed into UPPAAL templates and mission properties formalized into the UPPAAL timed automata language TCTL. We have evaluated our approach in three robotic missions and results show that the expected behaviour is correctly verified and the corresponding properties satisfied in the UPPAAL model checking tool.

Keywords: Formal Verification, Model checking, Multi-Robot Systems

Contents

List of Figures

ix

List of Tables

Acronyms

BIP (Behavior, Interaction, Priority).

CNL Configuration Network Language.

CRGM Contextual Runtime Goal Model.

CTL Computational Tree Logic.

DSL Domain-Specific Language.

FSA Finite State Automata.

FSM Finite State Machine.

GM Goal Model.

HDDL Hierarchical Domain Definition Language.

HRI Human-Robot Interaction.

HTN Hierarchical Task Network.

iHTN instantiated HTN.

LTL Linear Temporal Logic.

MCMAS Model Checker for Multi-Agent Systems.

MDE Model-Driven Engineering.

MPL Model-Based Processing Language.

MRS Multi-Robot System.

MutRoSe Multi-Robot systems mission Specification and decomposition.

NASA National Aeronautics and Space Administration.

NSHA Network of Stochastic and Hybrid Automata.

NTA Network of Timed Automata.

OBDD Ordered Binary Decision Diagrams.

PARS Process Algebra for Robot Schemas.

PCTL Probabilistic Computational Tree Logic.

PDDL Planning Domain Definition Language.

ROS Robot Operating System.

SAIT Samsung Advanced Institute of Technology.

SAS Self-Adaptive Systems.

SHR Samsung Home Robot.

SMC Statistical model checking.

TCTL Timed Computational Tree Logic.

TDL Task Description Language.

UML Unified Modeling Language.

VIPARS Verification in Process Algebra for Robot Schemas.

vTSL verifiable Task Specification Language.

WMTL Weigthed Metric Temporal Logic.

XML eXtensible Markup Language.

Chapter 1

Introduction

1.1 Motivation

The Multi-Robot System (MRS) field has grown significantly in the past few years. From task planning to control theory, this field holds many open challenges for researchers. Some of the main reasons for that are the increasing complexity of tasks entrusted to robots, robust collaboration between human and robots [6] and the need for unique domain-specific restrictions for verification and certification of safety-critical MRSs [7]. Some of those scenarios today include hospital robots [8], social robots [9] and robot assistants [10]. Many of these systems share the similarity of directly or indirectly interacting with humans during their operations, which, in turn, demand a more robust certification for their safety [11] and mission correctness. Therefore, it is imperative that robot systems must not contain any design flaws that could compromise the integrity of humans involved in their operation.

Model checking techniques are formal techniques for verification of a given model of a system through analysis of whether it satisfies specified properties or not [12]. The formal verification of systems offers automatic and exhaustive verification of the state space in finite state systems, assuring that any changes made to the specified model will not incur in new unforeseen errors. These specifications can be evaluated in terms of properties, such as safety, security, efficiency, reliability, dependability, etc. Model checking has been used extensively in the MRS field [13, 14, 15] as it is quite useful for evaluating if multi-robot models working in different settings are free of deadlocks and other design problems overlooked during design.

Since many robot systems have completely different context settings and objectives, their representation can be vastly different [16]. Therefore, several software engineering techniques are employed for designing robotic systems. Specifying behaviour can be done through frameworks, in fact, a lot of middleware architectures and Model-Driven

Engineering (MDE) techniques have gained traction for their ability to engineer a MRS with unique characteristics [17, 18]. Another famous approach is the use of graphical notations, which can be used to depict systems with a large set of parallel and/or sequential actions. The graphical notation is most useful for its inherent characteristic of visual representation, offering a common ground for both stakeholders and engineers to discuss specific implementation details with the aid of an illustrative system description. Some of the most known approaches are Finite State Machine (FSM)s and flowcharts such as RoboFlow [19]. On the other hand, one can also use Domain-Specific Language (DSL) approaches to represent a MRS with textual language. DSLs have two central characteristics: first, as the name suggests, their expressiveness must be directed to the specific domain, i.e. the use of a specific language must be justified by a significant gain in expressiveness during design. Second, the notation must be comprehensible for stakeholders while also being machine tractable [20]. Therefore, it is highly recommended that stakeholders decide which important features should be addressed in MRS due to scope restrictions in certain DSLs.

Another important concern is at what level of abstraction the specification must be, i.e. low-level specifications for MRSs would involve more detailed control over tasks, resulting in a larger system [16]. On the other hand, this approach would require more granularity and more thorough specification requirements for their inherent level of detail. Studies have shown that large systems are better suited for statistical verification, since other verification methods would often fail due to space state explosion errors [21]. Therefore, a high-level abstraction MRS is often recommended for non-statistical verification methods inside model checking. One other aspect that must be taken into consideration when designing a high-level specification is defining predicates: statements that may change during the course of a mission. They might be used to evaluate a certain universal state during the mission execution or simply checking if a robot state has changed while performing an action when it is supposed to. Likewise, it is possible to use agent capabilities working similarly as predicates to define if a certain agent has the capacity of carrying out certain actions.

There are many aspects when it comes to designing high-level MRS missions accurately. Some of them might be critical or not for mission success depending on the mission scope and its complexity. It is important to periodically submit a mission description to scrutiny (e.g. verification or testing) to ensure that all preliminary steps are being taken to guarantee mission correctness.

1.2   Context

An important aspect of the MRS mission specification is describing the system operation and its behaviour (also known as missions) [3]. Missions play an important role in defining main goals and tasks that must be carried out in order to achieve mission success. Furthermore, it is possible to create alternative mission paths should the main ones fail, this adds more complexity to the mission design overall but also expands the list of possible successful paths. Thus, regarding reachability, a mission is less prone to failure the more alternative mission paths available it has.

Mission requirements include movement and manipulation as robot capabilities, i.e. if a robot has some ability in order to carry out particular tasks. Robot capabilities are a way to define MRSs heterogeneity, i.e. if a group of robots differ from each other in terms of behaviour, equipment and abilities. Heterogeneity can make MRSs more complex as they grow larger in size [22].

Other mission requirements include: predicates or statements concerning the mission environment or the agents involved; and task ordering, as some tasks can be impossible to perform in a particular order if a previous requirement was not met e.g. a robot must pick a glass of water before delivering to its destination, this is usually considered under the communication aspects of systems, as they often need to coordinate actions with other robots in various missions.

Multi-Robot systems mission Specification and decomposition (MutRoSe) is a mission modelling framework for goal-oriented, high-level MRS specifications. It specialises in decomposing its input files into hierarchical task plans and outputting valid combinations of task instances as well as the execution constraints between them. In order to do so, it needs a GM [23, 24, 25] with domain-specific contextual runtime additions to accommodate flexible and real-world scenarios and a Hierarchical Domain Definition Language (HDDL) [26] file, which is responsible for describing hierarchical tasks pertinent to the mission domain.

Similarly to specifying MRSs, verification formalisms are also a very complex issue in MRS; it is possible to choose from a variety of different formal methods. Formal methods are mathematical techniques for specification and verification of properties in systems. They can be employed in MRS using formal verification tools for design, simulation, verification and testing. Besides, they offer potential for automation in software systems and MRS systems as well due to their re-usability feature. The survey in [16] identified and classified formalisms used in MRS, some examples are set-based (such as the B-Method [27]), state-transition systems [28] and temporal logics [29], for instance Linear Temporal Logic (LTL), Computational Tree Logic (CTL), Probabilistic Computational Tree Logic (PCTL) and Timed Computational Tree Logic (TCTL).

Among the verification tools, model checking is the most prominent and flexible verification approach due to its automatic nature and the ability to check for every combination of states within a model [16]; these characteristics also guarantee that an inexperienced user will be able to quickly design a specification then exhaustively check for safety, liveness and other properties within the model. This is not always true for other methods such as theorem proving or simulation [30] which may require additional specification (e.g. for the environment) for a thorough verification and a more skilled user beforehand. Within model checking, one can use one or more different formalisms to tackle a MRS design, this is mostly done by using process algebras or temporal logics.

One of the direct advantages of using verification is because it is an effective technique to outline potential design errors [12]. As shown in Fig 1.1, during a software lifecycle, errors detected during the conceptual design stage are about 40% less costly to fix compared to those detected in operation. Additionally, model checking verifies if important properties are maintained throughout system operation.

UPPAAL [5] is an integrated tool environment used for the creation, verification and validation of timed automata networks, a subset of FSA systems. UPPAAL has three main parts: a description language, a simulator and a model checker. These components will be outlined thoroughly on Section 2. While UPPAAL has a great focus on task synchronisation and model checking real-time systems (i.e. using TCTL), it can also be used to CTL as well by simply omitting the timed properties in a model. It uses locations as an abstraction for states and its transitions are defined by invariants, guards and synchronisation channels. UPPAAL has been used extensively to model and verify many MRSs [31, 32]. UPPAAL files are written in eXtensible Markup Language (XML).

1.3   Problem Definition

Demonstrating MRS specification correctness can be difficult without verification processes in place due to their complexity, multiple robots configurations and unknown context conditions, predicates, etc. might greatly increase the number of states inside a mission specification. Therefore, a verification technique such as model checking applied to MRSs specifications to identify potential inconsistencies would help mission designers to reason about mission specifications during early stages.

Thus, verification directly generated from specification models in high-level specification would impact positively on the accuracy of properties being evaluated. Other important challenge is accurately describing all important aspects of a high-level mission from the verification process, as other system properties may not be fully covered, even

if they are evaluated during verification. Defining the important aspects of a mission can be quite complex as it varies significantly from one mission specification to another.

In this work, important characteristics are defined as several properties such as reachability or mission correctness concerning predicates, capabilities and mission ordering which could be facilitated if identified through model checking and its exhaustive state space exploration. For instance, assume that a predicate p would drive the mission to failure every time it was set to true, hence indicating it must be either removed or safely guarded for certain contexts of operation in the mission specification. Depending on the mission complexity, the designer might not be able to identify this alone without a verification process in place.

This work aims to automate the verification process of high-level MRSs mission specifications. Specifications can range from behavior, planning, robot capabilities and coordination protocols between robots. This approach particularly focuses on MRS heterogeneous missions and how they can be verified through formal methods concerning the the correctness and consistency of MRS specification model and its requirements expressed in the form of temporal properties. In order to verify the MRS mission specifications, the generated models will be submitted to verification using the UPPAAL tool and their properties will be evaluated via TCTL formulas. UPPAAL was chosen for this work due to being able to represent a system as a Network of Timed Automata (NTA), extended with data types. It supports the system design as a collection of non-deterministic template with control structures able to communicate with each other through the use of channels or shared variables [33].

It is possible to evaluate MRS mission specification as verification properties as some works already show [34, 13]. Other works in MRS formal verification follow a similar workflow to provide a straightforward process when generating specification model then offering a verification technique for the given model in order to evaluate its correctness [35]. Therefore, an automated verification technique such as model checking applied to the specification of multi-robot models are able to provide more degrees of safety when compared to other verification techniques such as testing or simulation.

Concerning the properties that need verification, model checking already defines some default properties such as safety (something bad will never happen), liveness (something good will eventually happen), reliability, security, availability, survivability, maintainability, dependability and others. This work aims to assure safety and liveness inside a MRS specification, but also tries to guarantee mission reliability by ensuring to a certain level that they are correctly specified and able to potentially show the presence of design flaws in the MRS specification.

Although the mission describes the high-level tasks that the MRS must accomplish,

Figure 1.1. Software lifecycle and error introduction, detection and repair costs [1]

it is important to note that the mission specification must not necessarily explain how it will be achieved. Instead, it shows what tasks may be executed in order to successfully complete the mission [36]. In various MRS applications, this level of detail is crucial when the scope of the specification is still being defined, for it will define what properties are verifiable depending on the granularity of the system.

We should note that specification concerns such as mission layout (e.g. terrain characteristics, wall positioning, etc.), physical, kinetic or environment properties are out of this work's scope. Therefore, our verification process does not include robot implementation errors or mission environment problems due to the high-level perspective this work focuses on.

In order to be able to verify mission specifications automatically, the generation process must abide to rigid specification rules to attest that the output given by any of the specification files created will always be the same for a given input model. Thus, it is important to precisely outline how each member included in specification files relates to the verifiable model e.g. how a mission goal would be represented in the generated file and how the rule applied would be the same for every goal.

Robot swarms [34] are an example of homogeneous MRS due to no specialised robots. By specifying different capabilities as one of the many high-level mission requirements needed to be met by verification, it is possible to define if a predicate is fundamental for the achievement of a certain mission or what are the possible execution paths to achieve a certain goal. Which leads to the first research question:

6

> **Research Question 1. (RQ1) :** How to automatically verify mission specifications of heterogeneous MRS from a high-level perspective?

The second research question emerges from the fact that the generated verifiable files must retain important properties in order to assess the mission specification correctness. Thus, the scope for the following research question needs to be defined regarding the first one. For instance, if a given mission specification model is incorrectly specified, then the generated model verification must output some error indicating that the properties are not satisfied due to the inconsistency occurring in the model i.e. the properties specified must conform to the original model in a comprehensible manner. Furthermore, the error must relate to what problem exists in the specification and preferably suggest or give hints to what are the possible alternatives to fix them in a way to help the mission designer.

Some of the relevant properties MRS mission specifications verify are safety, security, correctness and others. As one might expect, it is important to assure to a certain level that mission correctness is achieved. Likewise, one can verify safety by ensuring absence of deadlocks. Other relevant characteristics such as reachability, i.e. being able to reach a certain path during the mission, or liveness are also possible inside verification through model checking.

The second research question aims to extract relevant characteristics as properties and other domain-specific MRS properties relevant to the mission context as well as verifiable in UPPAAL. One of its flaws is not allowing nested operators when writing formula queries, thus some properties are automatically ruled out by the verifier or require some modifications for further verification. Nonetheless, some characteristics must be addressed when it comes to fully verifying robotic mission specifications that are not common properties to all robot systems. For instance, if there is a mission path capable of accomplishing the mission with a certain set of capabilities enabled or if the needed preconditions are met before a certain goal or task. The relevant characteristics must be extracted from the specification model as verifiable properties in a comprehensible manner.

Another concern derived from the first question is the possible loss of meaning during the verification stage i.e. the specification and the verification model do not have the same implied properties or some properties are missing, and thus would render the verification model partially or completely useless. Therefore, both generation of verifiable files and verification properties processes must be sound and thoroughly specified to assure that such properties were not ignored during the generation process.

> **Research Question 2. (RQ2) :** Is it possible to extract relevant characteristics from MRS mission specification models as verifiable properties?

1.4   Contributions

The contributions for this work are twofold:

1. a verification process for high-level MRS mission specification to assure its correctness and identify potential inconsistencies early in the MRS mission engineering process. This is achieved through a strict set of mapping rules between mission specification and UPPAAL elements;

2. We also propose a framework that automatically implements this translation into UPPAAL models and properties. The output intended is as a set of verifiable TCTL properties and UPPAAL models generated from MRS mission specification inputs in the form of goal models and complex tasks expressed as Hierarchical Domain Definition Language (HDDL).

Additionally, a case study verifying mission scenarios from RoboMAX will be used for evaluation of this work. Figure 1.2 depicts the overview process for MutRoSe along with a proposed contribution. The area circled in red depicts the proposed addition to the current process. First, the mission specification elements are mapped and generated as a UPPAAL NTA, then the model is verified using UPPAAL model checker verifier tool. Should the specification verification be incorrect, the user is then able to correct the specification files and submit them once again for verification, restarting the process, it is important to stress that the restart is not automatic, however, given the arrow pointing back to mission specification files. It only points out that the same file (now corrected) is used once again as input. Note that the main contribution is an automated generation process derived from the models. One should note that the world knowledge is excluded from this verification process, that is due to the fact that the world knowledge if considered in this approach, would instantiate variables inside the verification model, this is not the best intended option since verification in UPPAAL is able to cover extensively multiple paths of execution. Therefore, the world knowledge is not an input for this verification process.

1.4.1   UPPAAL

UPPAAL is the model checking tool used in this project for specification and verification of MRSs. Its 3 parts (Design, simulation and verification) consist in an integrated envir-

Figure 1.2. Proposed contribution overview

onment that will be used for designing and verification of properties. It uses TCTL as formalism for verification. The designs are focused on channel communication between timed transitions, but the latter can be omitted by the user if the system does not contain any timed constraints. Additionally, UPPAAL verifies properties by using TCTL, likewise, timed constraints can be also be omitted, allowing the verification of non-timed properties as well.

UPPAAL is a tool used in several works in the verification field [37, 38], thus establishing its academical prominence, additionally, it provides a rich environment for verification of its models. It was the chosen tool due to its ability of providing a comprehensive model ordering through template graphs, moreover, its communication channels and variables are useful to link and describe many templates as an unique system.

Additionally, UPPAAL has many industrial case studies [39, 40], which proves its resourcefulness in both academic and business settings. This can be attributed to its responsive interactivity and friendly interface when designing templates. Arguably, UPPAAL has MDE features as it is able to break down complex systems in separate templates described as models, which helps to describe various systems timed scenarios.

1.5 Dissertation Outline

The remaining chapters of this document are structured as follows: Chapter 2 contains the relevant theoretical background. Chapter 3 presents the solution proposed in this approach. Chapter 4 displays experiments and their respective results, along with verification of properties. Chapter 5 approaches related works in MRS. Chapter 6 concludes this document with final remarks and directions for future works.

Chapter 2

Theoretical background

2.1 Goal Model

In requirements engineering, it is often beneficial to describe a system as a set of object-ives and the related steps towards their achievement. In goal-oriented approaches, goal models are a popular way to graphically describe a tree structure containing tasks and goals performed by certain actors in a bottom-up fashion. They also provide a comprehensive and intuitive language, which is useful for quick visualisation of high-level mission specifications.

In Fig 2.1, there is an example of a goal model. Goals are shaped as rectangular circles and the tasks are represented by hexagons. The set of goals and tasks refer to the actor responsible to enact them. The main task is the root node of the tree, if all sub-goals and tasks are performed accordingly, then the root goal will be achieved. Usually, a goal model has more than one way to achieve the main goal, justifying the need of a complex diagram to represent.

In order to further improve the representation of goal models, CRGM adds runtime annotations and contexts to the goal model. Contexts can be defined as a partial state of the system's surrounding world that may impact it negatively or positively. The algorithm which defines if the main goal is achieved, namely achievability [2], considers all possible path branches instances of contextual settings in order to satisfy the root goal, similar to the SAT problem. A similar process is done in CRGM missions by MutRoSe to derive all possible mission decompositions and how they can be achieved.

2.2 HDDL

Hierarchical Domain Definition Language (HDDL) is a language extension of Planning Domain Definition Language (PDDL) for hierarchical planning, the extension adds hier-

Figure 2.1. Goal model example for a museum's visitor assistance system [2]

archical planning characteristics while trying to preserve all other aspects of the original PDDL. The hierarchical language is responsible for representing a domain with abstract tasks and its respective methods. This domain may also contain variables and predicates related to them. A HDDL file may have the following elements:

- types: the list of types allowed for variables;

- constants: constants defined for the domain;

- predicates: the possible predicates (preconditions and effects). Predicates may act as constraints in the case of preconditions or as assignments in the case of effects;

- task: abstract task with name and parameters containing one or more methods;

- method: method with name, parameters and respective types, preconditions and subtasks;

- action: an atomic primitive task containing parameters, types and predicates

These elements are organised in tasks: they contain the different types involved in one or more methods that can execute the task. A method contains the actions that must be accomplished to finish the task and if their ordering is sequential or parallel. Additionally, methods may have preconditions defined by predicates, which could constrain the execution of the method due to preconditions not being met. Actions have parameters containing the types involved, since this is done in an hierachical manner, the types involved in an action also belong to the method. Actions also contain effects: they work as statements which may update values of predicates in HDDL.

2.3 MutRoSe

MutRoSe [3] is a framework for hierarchical task planning with strict rules for system description and world knowledge. Additionally, the project contains examples to help beginners to understand the tool and design their own mission specifications and output their tasks decomposition provided that mission specifications and world knowledge are made correctly. The output for MutRoSe are instantiated HTN (iHTN)s, which are the valid mission decompositions based on specification constraints, also known as mission plans. Hierarchical Task Network (HTN)s are task networks that represent possible decompositions given a HDDL specification and differ from iHTNs for their lack of concrete variables instantiated. Thus, iHTNs are concrete instances of previously decomposed HTNs inside MutRoSe. In other words, Multi-Robot systems mission Specification and decomposition (MutRoSe) is a goal-oriented DSL framework used to specify multi-robot mission plans. MutRoSe is concerned with the high-level task planning of multi-robot missions and the allowed decompositions available given a specific state of the system and its environment. After given the mission specification files, it runs an algorithm and derives the valid mission decompositions as output.

An incorrect specification can compromise the entire decomposition process. The reason is that MutRoSe cannot detect if a mission has valid decompositions up until its execution, leaving the mission planner to discover what is the model error without any assistance. Moreover, there is not a generation process for MutRoSe missions as verifiable specification files. This process should be done automatically for valid MutRoSe mission specifications, i.e. a specification syntactically correct, but not necessarily semantically correct, as it could contain design errors. Therefore, model checking could be greatly beneficial to MutRoSe specification files as they are not subjected to any verification techniques and these errors could impact a MRS mission performance or even its achievement. Figure 2.2 shows MutRoSe process overview

2.4 The UPPAAL Model Checking Tool

Model checking is a formal verification method that "explores all possible system states in a brute-force manner" [12] and can help to verify systems at an early stage of design. A popular model checker to verify real-time systems is UPPAAL [5]. It is used for the creation, verification and validation of networks of timed-automata (NTA), a subset of FSA systems.

UPPAAL provides a graphical interface divided into three main parts: the editor, the simulator, and the verifier [5]. In the editor, systems are modeled as networks of timed-

Figure 2.2. MutRoSe process overview [3]

automata inside template files. These networks are composed of locations connected by edges that can execute functions, hold logical conditions, and synchronize with other automata in the system through channels [41]. UPPAAL uses locations as an abstraction for states and its transitions are defined by invariants, guards and synchronisation channels. UPPAAL has been used extensively to model and verify many MRSs [31, 32]. Finally, the system defined in the editor can be executed in the simulator, which displays the state of the automaton at every step.

Table 2.1. Types of TCTL formulae supported by UPPAAL [5].

TCTL formula	UPPAAL formula	Description
$AG\ \varphi$	A[] φ	φ should be true in all reachable states, i.e., for all paths φ is always true.
$EG\ \varphi$	E[] φ	The should exist a maximal path for which φ is always true, i.e., in every state of this path.
$AF\ \varphi$	A<> φ	For all paths, φ should be eventually true.
$EF\ \varphi$	E<> φ	There should exist at least one path, for which φ is eventually true.
$AG(\varphi \to AF\ \psi)$	φ -> ψ	For all reachable states, whenever φ is true, then eventually ψ will be true.

According to several definitions in [5, 42, 43], a timed automaton is defined as a tuple $(L, l_0, C,$
$A, E, I)$ where L is the set of available locations, $l_0 \in L$ is the initial location, C is the set of clocks, A is the set of actions, co-actions and the internal τ-action, $E \subseteq$

14

$L \times \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with an action, a guard and a set of clocks to be reset, and $I : L \rightarrow B(C)$ assigns invariant to locations. A NTA is therefore, a network of n timed automata $Ai = (L_i, l_0^i, C, A, E_i, I_i)$. Since no clock constraints are used in this generation (as MutRoSe itself does not contain timed constraint properties), $C = \emptyset$. Templates automata are defined with a set of particular parameters defined in our approach by the HDDL types used during task execution, these parameters may be passed by value or by reference. Due to flexibility concerns, this work uses pass by reference to define which variables will be passed as parameters.

Properties in UPPAAL are specified in Timed Computational Tree Logic (TCTL) language [5], which has its syntax shown in Table 2.1. As TCTL implies, UPPAAL supports verification of timed automata, such as real-time systems. Nevertheless, it can be used for verifying untimed software by simply omitting the timed properties in a model[44].

Chapter 3

Proposed solution

This chapter contains a detailed explanation concerning the proposed solution discussed in Section 1.3, comprising the stages of development necessary to achieve the solution. This section is organised as follows: first, it will be discussed the overall proposed solution, with a descriptive image showing what the intended contribution is. Next, another figure will depict in details the process overview used in this work. The process is divided in stages and the following sections are defined by each stage described in the figure. For instance, the generation stage will cover the mapping rules used to map MutRoSe elements to Uppaal structures, alongside a general overview of how the main components of the NTA interact. Finally, a more internal view of the parsing and generation process is depicted in order to give the reader a more concrete sense of what is happening inside the automated process.

3.1 Process overview

The process uses MutRoSe execution to perform the creation of output files used for this approach, from then on, it is in a separate program used for parsing and generation. As of now, the verification process is not fully integrated with MutRoSe, as Figure 1.2 suggests, but it is possible to generate Uppaal models by executing MutRoSe and then the program with the output files.

An explanation of the process itself is available in Figure 3.1, which depicts the input files and processes involved in the parsing and generation of Uppaal models. The process begins by executing the MutRoSe framework with input files derived from the specification files, namely, the MutRoSe execution stage. Next, the generated files are used as input for the parsing stage, where they are parsed as data structures to be used in the generation stage. Generation comprises the generation of domain, goal model templates and verification queries. Lastly, the verification stage is responsible for evaluating TCTL

Figure 3.1. Process overview

queries designed to verify mission properties. As indicated in Figure 3.1, we further delve into the sub-parts of our process in the forthcoming sections.

3.2 MutRoSe execution stage and parsing stage

The execution of this stage is necessary to extract information to parse it into data structures afterwards during the parsing stage. The parsing stage is basically responsible of reading and transforming the generated files in data structures responsible for the actual generation process. During the execution stage, two main files are generated from the goal model file and three from the domain definition input file. For the goal model, these files are the goal nodes info file and the goal model order file. The goal nodes info contains all information concerning a node (i.e. a task or a goal) inside the GM.

As for the domain definition, the main generated files are: the types and variables information file, the available methods for abstract tasks and the method ordering file. The first one contains the listed variables in the HDDL file and their respective types. Next, the available methods for an abstract task file contains the names of one or more methods available in the domain definition. Lastly, the method orderings contains all possible orderings for actions within a method.

Examples of generation files are shown in A.1 for both domains (i.e. GM and HDDL). In the following sections, we will discuss the generation stage and the verification stage in a high-level fashion, i.e. the sections will not concentrate on specifics of code. The

Figure 3.2. Goal model example
]

generation stage section will also contain the mapping rules needed to generate Uppaal templates and additional structures derived from MutRoSe elements.

3.3 Generation stage

The generation stage mainly consists in compiling the information available in the parsed data structures and translating them to templates inside Uppaal. The already parsed data structures are sent to this stage where they are submitted divided into two main processes: generation of domain methods templates and generation of goal model templates. The generation of domain methods is derived from files related to the HDDL while the goal model templates derive from mission ordering and general goal model information data structures. Both processes also comprise the global and system declarations (textual structures) used for the templates. After the generation of templates, templates are merged into the same NTA and some automatic verification queries such as deadlock freedom are added to the verification queries automatically, since they follow the same syntax in every NTA.

In order to do so, a strict translation process must be established to determine how the elements of specification in MutRoSe will be adapted to a generated Uppaal NTA for verification while preserving the original semantics. Therefore, it is imperative to display in a subsection, namely mapping rules section, to describe exactly how this process occurs.

Additionally, following subsections will also contain specifics of the generation process itself with a breakdown of how mapping rule is applied during the generation.

3.4 Mapping rules

To generate a coherent generation, applicable to all missions designed in MutRoSe, one must define how elements present in the original specification are translated to a verification grammar (i.e. the UPPAAL NTA). Table 3.1 express the rules derived from elements which are described in the GM or the HDDL input files and how they are created within the generation process for the NTA. In addition, rules will be further elaborated in their respective subsections. A UPPAAL timed automaton is defined as a non-deterministic finite state machine enhanced with clock variables where the clock variables are evaluated to real numbers during simulation. In the next subsection, we will use the semantics of the definition present in [43, 5] as grounds to establish the generation process, this semantics will be used throughout this section.

NTA generation

Two main automata generated are defined as the goal model level template and the task level template, note that templates and automata will be used interchangeably from now on. The goal model level template is one automaton responsible for coordinating task and method execution in the order defined by the CRGM tree, whereas the task level template is a collection of m available task methods and templates responsible for execution of the subtasks needed to achieve a particular abstract task, defined in the HDDL file.

When mentioning certain MutRoSe elements, it is worth noting that there is an input file responsible for each rule ID. For instance, consider rule #1: for the goal model level template, no particular types are necessary for its creation, therefore no parameters are used in this template by default, while the task level template may use one or more types, depending on the types used in the actions defined in their subtasks. Both levels have their declarations stated in the global declarations, which, as the name suggests, is visible to all other templates. It is beneficial for tasks to be able to check each other status during mission simulation, such as capabilities, which are globally visible. This is justified by the fact that types are elements originated from the HDDL inside MutRoSe. The following rules try to divide template responsibilities in order to clarify the generation process, however, this is not possible at all times, since some interaction is needed for both levels to cooperate inside the same network of automata.

The common flow between those two automata is as follows: the goal model template triggers the execution of goals and tasks as described by the goal model input file, goals

19

may have runtime annotations which are critical to mission ordering, while tasks are used as execution placeholders to their respective methods. Whenever a task is executed, the goal model then triggers a channel to execute the particular method template for that task. The method may finish with a successful or failure state, this indicates that the task has finished in both cases. Next, a channel is triggered by the task method warning the goal model template that its execution has ended, which delegates the simulation execution back to the goal model level. This is done until the mission is finished or fails by being unable to execute one or more tasks.

Therefore, one of the immediate advantages of using a verifiable model is to investigate execution traces and how predicates or other mission parameters such as variables may impact on their behaviour. Next subsections dwell deeper in how rules interact during the model generation and how these constructions are helpful during mission simulation and/or verification.

Rules #1 and #2

Types in HDDL are used to define allowed types for variables in the domain [3]. Types may have predicates, which are more thoroughly defined in rule #3. In our generation process, a type is mapped as a struct type with a particular method and variables are instantiated according to the maximum number of parameter variables present in one single task. Assuring that the number of instance variables will suffice the required amount of variables associated with that type for the mission description.

A type is therefore a set of predicates $T = [P]$ where $VP \subseteq P$ is the subset of valid predicates in P. As rule #2 states: types without preconditions or effects present in the domain file (i.e. valid predicates) are discarded, as they are not present in the domain definition. This is done inside the generation process by evaluating the available methods, their subtasks and actions and removing the types without valid predicates until only VP are mapped in our approach. In MutRoSe semantics, types can also have their types defined through the world knowledge, a secondary file which contains objects that will replace variables with instances. In addition, the world knowledge contains definitions of predicates and functions being initialised. Since the world knowledge is being discarded for the sake of generality, some variables have no defined value and cannot be properly taken into account without this file.

Rules #3, #4 and #5

Predicates are defined as boolean expressions which can be used as preconditions or effects and are always defined inside a type. Consider the equation with the following semantic

of a transition [42]:

$$l \xrightarrow{g} l', \text{ where}$$
$$g = t.precondition == true$$

(3.1)

Equation 3.1 defines a transition from location l to l' bounded by an guard g, which means that the transition will only occur when the *t.precondition* is true. In this approach, the start location is denoted by l of a method with a predicate *precondition == true* of variable t from a type *Type*. The Figure in rule #4 row depicts a similar transition to an action bounded by the same guard where l named as "action" for clarity purpose. In other words, the action will only be performed if the precondition stands, as defined in the domain specification.

This, however, raises a problem with preconditions defined as guards: if the precondition is not met by some reason, this would result in a deadlock inside the model, as there would be no other transition available for the template to go to. This was solved in this approach by adding an extra location with two new transitions: one containing a guard with negation of the predicate as shown in rule #5 to avoid deadlocks; the other transition goes back to the initial node, triggering method failure with the assignment of a boolean variable to true (namely method_0_failed) which denotes mission failure in templates. The transitions are both represented in the Figure of rule #5 and in the equation below:

$$l \xrightarrow{\neg g} l_{fail},$$
$$l_{fail} \xrightarrow{u} l, \text{ where}$$
$$\neg g = t.precondition == false \text{ (i.e. the negation of } g\text{)},$$
$$u = method\_0\_failed = true$$

(3.2)

Where l_{fail} is the additional location created for failure and l remains the same location from Equation 3.1, stressing that both must stem from the same initial location where the precondition rule appears in order to prevent a deadlock condition. $\neg g$ is the negation of the precondition generated simultaneously. In the case of having more than one predicate in the same transition, UPPAAL is able to support n predicate clauses using boolean algebra: consider P and $\neg P$ the set of n predicates in a transition, thus the following equation depicts how predicates and their respective negations are generated:

$$P = p_1 \wedge p_2 \wedge p_3 \wedge ... \wedge p_n$$
$$\neg P = \neg p_1 \vee \neg p_2 \vee \neg p_3 \vee ... \vee \neg p_n$$

(3.3)

Where $p_1, p_2, ..., p_n$ as well as their negated counterparts correspond to individual predicates, such as g and $\neg g$ in Equation 3.1 and 3.2. It is also possible to note that

synchronisation issues are addressed by communication channels. While there are not imperative mapping rules for them as they are not derived from MutRoSe elements, they are present throughout implementation in order to guarantee execution in the correct order of the NTA methods defined by the goal model template, which will be explained in rules destined for the GM input file.

Rule #6

Predicates also come in the form of effects, which can be defined as the triggered predicate after performing an action (i.e. a transition). Likewise, a similar pattern is found in rule #6, where instead of being a guard, it takes form of a UPPAAL update. Updates are used in UPPAAL to assign values to variables or invoke functions defined in declaration templates. An update transition works similarly, where instead of being the target location for a transition, it is its source location. However, they do not require a negation nor extra transitions as preconditions do, this is due to the fact that they are only an assignment to a variable which side effect is changing the system state, thus, they do not cause any deadlocks. Referring to the rule #6 Figure in Table 3.1, an equation below depicts how an effect could be generically expressed:

$$l' \xrightarrow{e} l'', \text{ where}$$
$$e = t.effect = true$$
(3.4)

Where $t.effect$ is another predicate from the same type struct variable t, location l' is the source location and l'' is the end node if the method does not contain any more subtasks or a subsequent action. For reference, an example of the struct used can be seem in Figures of rule #1 and #3.

Rules #7 and #8

Capabilities are one of MutRoSe particular additions to HDDL syntax and are used to define capabilities necessary for mission achievement. As such, they work in a similar manner as predicates, with the exception that capabilities are not assigned such as in rule #6.

Capabilities have a global scope when mapped to UPPAAL as boolean variables but do not possess any types and are individual instances. This however poses a limitation to how these capabilities are used inside UPPAAL, since they are converted directly to a variable during generation, it is not possible to have multiple instances of a given capability, whereas predicates may have as many variables as possible. Capabilities are mapped as such mostly because it is not possible to infer how many capabilities will be

needed using only the domain file. The following equation depicts the original capability transition followed by the additional transitions and location added to prevent deadlocks:

$$l \xrightarrow{c} l', \text{ where}$$
$$c = capability == true$$
$$l \xrightarrow{\neg c} l_{failc},$$
$$l_{failc} \xrightarrow{u} l, \text{ where}$$
$$\neg c = capability == false$$
$$u = method\_0\_failed = true$$

(3.5)

It is important to stress that while Equation 3.5 is very similar to equations regarding preconditions (i.e. Equations 3.1, 3.2) l and l' are different locations from the former equations used here for clarity purposes. Furthermore, it is possible to define a set of C capabilities for a given transition in which the generation process for l, l', l_{failc} would behave very similarly as Equation 3.3. Lastly, capabilities too might compromise the task execution, therefore its transition also contains the update u.

Rules #9 and #10

```
1  (: task AbstractTask : parameters (?r1 ?r2 - robot ?p - person))
2      (: method method -0
3                  : parameters (?r1 ?r2 - robot ?p - person)
4                  : task (AbstractTask ?r1 ?r2 ?p)
5                  : precondition (and
6                          (precondition ?r2)
7                  )
8                  : ordered - subtasks (and
9                          (action -0 ?r1 ?p)
10                         (action -1 ?r1 ?p)
11                         (action -2 ?r1 ?p)
12                  )
13          )
14      (: method method -1
15                  : parameters (?r1 ?r2 - robot ?p - person)
16                  : task (AbstractTask ?r1 ?r2 ?p)
17                  : ordered - subtasks (and
18                          (action -3 ?r2 ?p)
19                          (action -4 ?r2 ?r1)
```

```
20                      ( AbstractTask -2  ?r2  ?r1 )
21                   )
22             )
```

Listing 3.1. AbstractTask definition from domain file

Abstract tasks are used in HDDL to describe how they are achieved through the execution of a method m contained in a set of methods M, which may contain sub-actions and sub-methods. The domain file does not contain explicit instructions of which methods will be needed for a particular mission setting, in fact, the method might not be used at all for that MutRoSe instance should it not be included in M. Thus, the generation process adopts the naive approach of generating all method templates. The generation process adopts this behaviour since the abstract tasks which will be executed are only known during the generation of the goal model template, where goal tasks are directly related to abstract tasks from the HDDL file. Thus, it is safe to conclude that the collection of UPPAAL template graphs related to a abstract task directly represents the said task.

In order to illustrate how the generation of task in HDDL to a UPPAAL template is done, suppose we have an abstract task with two methods as in Listing 3.1. It depicts an example of a HDDL abstract task composed by two methods, which are related to the task due to the task attribute (lines 4 and 16). method-1 does not contain a precondition while method-0 does (lines 5 through 7). method-1 contains an abstract task in its subtasks. HDDL specification supports nested abstract tasks inside other tasks, the solution adopted in this work is to use yet another synchronisation channel inside the method template referring to the respective available methods for the abstract task in question. In an UPPAAL template, this means that there will be a transition channel linking the generated template of method-1 to the available methods of AbstractTask-2 when transitioning from action-4. Suppose that the only available method to execute AbstractTask-2 is method_2 (since its definition is not shown in Listing 3.1). Whenever the task method ends (succesfully or not), a channel triggered returns the simulation to the method. From then on, there are two transitions from which the method continues its execution, one is the remaining subtasks, where the underlying method has not failed and other where it has. For the failed method transition, there is a specific location (namely failed_AT) where the failure state is triggered, which has a transition going back to the end-method node, which triggers the channel indicating that the method has ended. Figure 3.3 illustrates how the following output would be for this method. It is important to stress that the only available method for AbstractTask-2 was method_2, thus, the synchronisation channels used in this example coincide with the specification. If there was more than one method for AbstractTask-2 to be achieved, this method would be included as an available transition
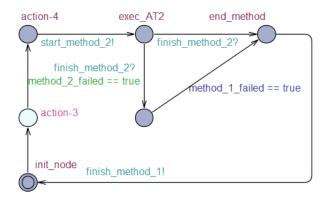
Figure 3.3. UPPAAL generated template for method-1

as well. Figure 3.4 displays an example for nested abstract tasks with two available methods.

Abstract tasks and methods coincidentally have parameters, which are used to define which parameter variables are used in their subtasks. Thus, the parameter generation derives from the domain file specification. One important exception is that if the type is removed due to not having valid predicates (as mentioned in 3.4), the type itself will be removed from the parameters list. As mentioned before, the parameters are defined by reference for two main reasons: one is that the domain file also does not instantiate variables, only defines which variables are used, thus it is possible to infer that the definition uses call by reference in the domain file as well. The second reason is that by adopting the call by reference approach when generating, it is possible for the end user to define which variables are used for each method in system declarations. It is possible to identify the parameters from the domain file in lines 15 and 3 in Listing 3.1, derived from the parameters needed for the task (line 1).

Both tasks and parameters are directly involved in system declarations. UPPAAL uses system declarations to define which templates will be instantiated as processes in that system instance. In more concrete terms, if a template is not attached to the system process, it will not be accounted for in simulation and verification stages. This allows for more flexibility while using the templates as the end user is also able to define which methods will be truly used in its system. For this generation approach, all methods are included in the system declarations. In addition, variables of a same type can be switched to evaluate new system configurations, this essentially means that if a variable r of type *Robot* is defined in the template, that variable may be reassigned in system declarations to another robot $r2$. In doing so, the end user may analyse the behaviour of a single robot throughout the entire mission to see if the mission itself is compromised somehow. The only pitfall for this approach is assigning variables not declared in the global declarations, which will obviously output an error.
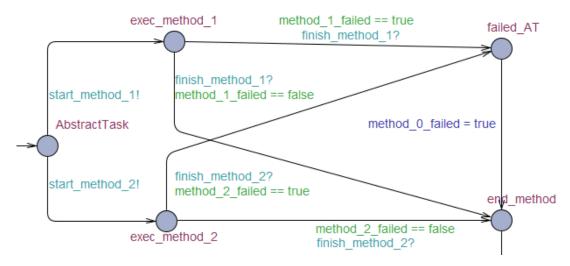
25

Figure 3.4. Method template with nested abstract task with two methods in UPPAAL

Rule #11

Actions (also known as primitive tasks) [3] are concrete tasks from the domain file which belong to one or more methods and need to be carried out to achieve a certain task. Actions may have preconditions, effects and parameters, alongside their types (type instances needed for that action to occur).

Aside from being mapped as locations and having transitions originating from or to them with guards or updates, actions themselves do not hold much importance since they do not go into details as how they are achieved. The reason is that actions should not be specific by design, which overall contributes to the high-level approach MutRoSe has.

Rule #12

In a GM, a goal represents an objective achieved by carrying out its sub-goals and sub-tasks. It is therefore the representation of a mission goal that is relevant to the mission context. MutRoSe adds another layer for goals when adding runtime annotations that may affect the order as well. The tree traversal in a goal model is done depth-first from the leftmost position, also known as preorder traversal. This order can be changed if a runtime operation takes place.

In UPPAAL NTA generation, Goals are the primary generated structure from the goal model level template. As stated before, the goal model level template consists of one template which replicates the ordering present in the CRGM file. Goals without runtime operators are only added to the UPPAAL template graph if they contain a leaf node containing a task in their traversal path, otherwise they are not generated. This is done to reduce the state space complexity without loss of meaning for both the model

and MutRoSe specification as the actual execution is carried out by tasks, there is not an issue in ignoring nodes which are not crucial for task achievement. Other goals that possess runtime annotations will be discussed in other specific rules.

Rules #13 and #14

Tasks in the GM translate to abstract tasks (domain file) by name, which, in turn, represent one or more methods. Tasks are only descriptions of which steps must be taken in a goal-oriented setting to achieve a particular objective, tasks only contain one id (e.g. $AT1, AT2, ..., AT_n$), namely task_ID, and a name which refers to the abstract task method name.

In the generation process, whenever a task node is encountered, the goal model level template creates two locations: one is the initial task location, named $exec\_[task\_ID]$ and other is the end task location, named $finish\_[task\_ID]$. The initial task location is responsible for being a transition target (i.e. an edge with an arrow pointed to in the initial task location) for a synchronisation channel where it triggers the execution of the method. The goal model level template is then halted at this location because the next transition to the end task location contains a synchronisation channel waiting for the task to be finished, thus it must wait for the channel trigger. The end task location is responsible for analysing the result of the task execution after its end was triggered and taking the correct deterministic transition afterwards. Similarly with preconditions, where there is a failure and a successful state, the end task location has two branching transitions to decide if the task has failed or not. This is decided by the triggering of the previously discussed variables in guards which denote mission failure for a method. Should the task fail and not inside a fallback operator, then this means that the mission has failed and the execution stops abruptly followed by the triggering of a variable which represents mission failure, named $mission\_failed$. Otherwise, the mission continues to the next locations or to the location representing the end of the mission. Figures in rules #13 and #14 depict how this pattern occurs in the goal model level template.

Rules #15 and #16

A fallback operator is a GM runtime annotation operator contained in goals inside the CRGM. If a goal contains this operator, a very specific pattern both in MutRoSe and in the generation process occurs. First, the rule for the fallback operator will be briefly discussed, next, the generation rule will be explained to establish the relationship between both representations.

A fallback runtime operator is one of the three runtime operators in MutRoSe. Having a fallback annotation means that the goal has an alternative course of action should the first one fail. The semantics for the fallback operator is:

$$FALLBACK(N1, N2) \tag{3.6}$$

Where $N1$ and $N2$ are the first and second id node and may be a task or a goal inside the goal model. What the fallback operator essentially does is: Should $N1"$ fail its execution, then $N2$ must execute correctly, or else the mission fails. The fallback operator has nodes $N1$ and $N2$ as children and its execution pattern differs greatly from others. For instance, if $N1$ finishes successfully, then $N2$ is not even executed. On the other hand, $N2$ should only be executed when a failure of $N1$ is confirmed.

In UPPAAL, the generation rule takes into account all three possible outcomes.

- If the first operand from fallback is successfully executed, then it transitions directly for the next node available (i.e. the sibling node, if it exists) or;

- if the first one fails, then the second operand is executed. If it also finishes with a failure state, then it diverges to a failed mission state;

- If the first one fails and the second one is executed successfully, then a transition is made where to the next mission node available.

This is illustrated by Figure 3.5 where we have the generation of a fallback operator as part of a UPPAAL template in the following syntax: $FALLBACK(AT1, AT2)$. $goal\_G[previous]$ is the goal location where the pattern begins, as stated in rule #9 and #10, it is possible to see the transition with a synchronisation channel triggering the execution of the $AT1$ task, executed by the $method\_0$ template. Next, in the $finish\_AT1$ task, there are two transitions: one to the next goal $goal\_G[next]$ and other in the case the method fails. In the failed method transition, it is possible to observe that the second task $AT2$ begins its execution, following the same pattern. After trying again with a different task, the pattern ends in a successful state or a mission failed state, represented by $missionFailed$ location, if both tasks should fail.

Lastly, another modification is made inside methods involved in fallback operands, stated by rule #16: if a template method is inside a fallback operator, a default failure location is added to it. This is done to assure that all mission paths allowed are explored, even if the method does not possess failure states defined by other conditions, such as abstract tasks failing or preconditions or capabilities not being met.

Rule #17

A sequential operator is a runtime operator in the GM inside MutRoSe. It is a very straightforward pattern: whenever a goal contains a sequential operator, all operands (i.e. goals or tasks) involved must be executed in that strict order, establishing an execution constraint. As opposed to a fallback operator, a sequential operator may have two or more operands, while the fallback operator is binary.

In UPPAAL generation, this is done by an algorithm which "unwinds" the goal model from the sequential root whenever a sequential operator is found. Unwinding the sequential root means that another generation process takes place to ensure that the tasks are sequentially executed in the order stated by the operator. The result for one task is depicted in the Figure in rule #17. The sequential pattern can be extended to one or more tasks,

Rules #18, #19 and #20

The rules #18 and #19 state that all generated NTA models possess boolean variables used to indicate whether a mission has failed or not in the goal model level template. Necessarily, one of them receives a true value after the end of an execution due to the fact that they are linked to locations situated at the end of the template graph or in failure locations. This value is used afterwards during simulations and verification queries to assert if a mission has ended successfully given a certain configuration.

After a mission has ended, it goes back to the initial node (beginMissionNode), where it can begin its execution again. Since the values are still stored, the $startMission()$ global function is used to flush these values whenever a new mission begins, this is done in the first transition of the system.

Rules #21 and #22

The initial nodes in templates play a central role in triggering mission or method execution but also pointing out that they have finished. In the goal model level template, aside from starting the mission, the beginMissionNode is also responsible for being the location where all final states concerning the previously executed mission can be seem during simulation.

As for the task level template, the init_node location is used to trigger execution of the method, while the end_method is responsible for triggering the synchronisation channel which warns the goal model level of its end. Both are generated for every NTA and are used during generation process by linking of the dynamic parts of the template (i.e. the mission specification).

Input file	Rule ID	MutRoSe element	mapped in UPPAAL as	Visual or Textual Representation
| HDDL | #1 | Types | Structs inside the global declaration if they have predicates related to methods used within the mission | ```
typedef struct {
bool effect;
bool precondition;
} Type;
Type t = {false, false};
``` |
| HDDL | #2 | None | Types without valid predicates (i.e. predicates not used as precondition or effects) are ignored in the specification | Not applicable |
| HDDL | #3 | Predicates | Boolean variables inside their struct types which denote the predicate value for that instance. | ```
typedef struct {
bool effect;
bool precondition;
} Type;
``` |
| HDDL | #4 | Preconditions (Predicates) | Transition guards in template graphs defined by the HDDL task description |
action
t.precondition == true |

Table 3.1 continued from previous page

Input file	Rule ID	MutRoSe element	mapped in UPPAAL as	Visual or Textual Representation
HDDL	#5	None	A new location and additional transitions are added for the negation of the guard in order to avoid deadlocks, if a predicate fails, the method itself fails and the task triggers its failure channel.task ends prematurely	
HDDL	#6	Effects (Predicates)	Transition updates in template graphs defined by the HDDL task description which assigns a boolean value inside a struct variable	

Table 3.1 continued from previous page

Input file	Rule ID	MutRoSe element	mapped in UPPAAL as	Visual or Textual Representation
HDDL	#7	Capabilities	Boolean variables without struct types which denote the capability value for that instance. For the template graph, they are used as guard conditions in actions with required capabilities	
HDDL	#8	None	A new location and two additional transitions are added for the negation of the guard condition in order to avoid deadlocks, if a capability fails, the method itself fails and the task triggers its failure channel. The method ends prematurely.	
HDDL	#9	Tasks	A collection of UPPAAL graphs containing one or more methods related to that task	Not applicable

Table 3.1 continued from previous page

Input file	Rule ID	MutRoSe element	mapped in UPPAAL as	Visual or Textual Representation
HDDL	#10	Task parameters	Called by reference as types displayed in the specification	
HDDL	#11	Actions	An atomic UPPAAL location for each action	
GM	#12	Goal	If a goal is within the subset of nodes (i.e. sub-goals or sub-tasks) that contain a task as a leaf node, this goal is included as a location in the goal model template	

Table 3.1 continued from previous page

Input file	Rule ID	MutRoSe element	mapped in UPPAAL as	Visual or Textual Representation
GM	#13	Task	Two subsequent locations are added, one triggers the channel execution for the one or more methods available for that task. The second one deals with the end of task execution and checks if the task has failed, and checks if the task has failed, depending on the task parent operations, this may trigger mission failure inside the goal model template	
GM	#14	None	A transition activating the method boolean variable indicating method failure is added to the goal model template. The mission fails if the task does not belong to a fallback runtime operator, where it may have an alternative task to execute afterwards.	

Table 3.1 continued from previous page

Input file	Rule ID	MutRoSe element	mapped in UPPAAL as	Visual or Textual Representation
GM	#15	Fallback runtime operator	Locations with additional transitions. If the first operand finishes succesfully, a transition links the last node of the first operand to the next sibling (i.e. the next task) or the end of the goal model template. If not, it is directly linked to the second fallback operator, where it triggers its execution. If the second operator also fails, the transition then goes to a mission failure state, ending the mission	See Figure 3.5
GM / HDDL	#16	Task / Fallback	Whenever a GM task is inside a fallback operand (i.e. being a child node), an additional failure location and its respective transitions are added by default in the method(s) template graph due to the specification stating that the particular method(s) may fail	

Table 3.1 continued from previous page

Input file	Rule ID	MutRoSe element	mapped in UPPAAL as	Visual or Textual Representation
GM	#17	Sequential runtime operator	Whenever a GM task is inside a sequential operator (i.e. it is a child node of the sequential operator), it is generated and executed strictly in the sequential order to prevent specification violations.	
GM	#18	None	A mission succesful node is added alongside a global boolean variable which denotes mission success	
GM	#19	None	A mission failure location is added alongside a global boolean variable which denotes mission failure	
GM	#20	None	A function named startMission() containing all global variables and struct variables being reset to false as mission starts so that no previous values are carried out to a new mission execution. They may be customised by the end-user to test new mission configurations	

Table 3.1 continued from previous page

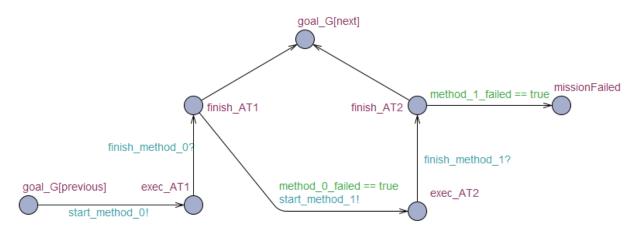| Input file	Rule ID	MutRoSe element	mapped in UPPAAL as	Visual or Textual Representation
GM	#21	None	In the goal model level template, a initial node is always created to denote the beginning of a new mission structure. This node is called beginMissionNode	
HDDL	#22	None	In each method template from the task level there is a initial location called init_node and another one called end_method. These locations are used to trigger the start and finish of method executions, respectively	

Table 3.1. Mapping rules

Figure 3.5. Fallback runtime operator template pattern

3.4.1 Generation of TCTL verification properties

Many of UPPAAL TCTL verification queries properties could not be automatically generated for some cases as they are somehow dependent of the generation process itself. However, some properties were possible to generate automatically since their syntax would not change from model to model and thus the generation was possible.

Some examples of automatically generated properties are deadlock freedom and reachability, which is described as whether the mission root goal will eventually be successful, this is also done with intermediary goals to show that ordering constraints still influence in partial mission achievement. All properties are described in Table 3.2, where each row represents a different property evaluated for this work: reachability evaluates if a mission can achieve its root goal given the correct configuration; mission ordering correctness evaluates if a certain goal is achieved after the execution of its task methods, used in this work to depict that mission ordering follow the same as the goal model, even sharing the same mission constraints; predicate or capability reachability is used to verify if a predicate or a capability with a certain value (i.e. true or false) might compromise the execution of a method or the entire mission as well, the example for this row contains a TCTL query where the left side of the formula is a capability and the right side is the variable triggered if a particular method fails; last property states that the system is free from deadlocks.

3.5 Verification stage

The verification of TCTL mission properties is done after the generation using the already completed NTA. Due to some of properties being boolean variables, it is also possible to explore other mission configurations by changing predicates and capabilities. Additionally, it is possible to test multiple configurations with different robots, this can be done by

Property	Description	Example
Reachability	If a root goal will be achieved successfully or not	E<>mission_complete
Mission ordering correctness or goal satisfiability	A goal is only reached if previous task methods are completed correctly	A[ ] var_goal_model_template.goal_G8 imply (not pickup_with_door_opening_0_failed or not pickup_without_door_opening_0_failed)
Predicate or capability reachability	A predicate and/or capability leads eventually to a failure state in a method	not manipulation - ->fetch_deliver_0_failed
Deadlock freedom	The system contains no deadlocks	A[ ] not deadlock

Table 3.2. Properties verified in missions

changing system or global declarations depending on which one the end-user plans to analyse. Once the model is completed after the generation, the verifier is used to assert verification queries written in TCTL. One limitation is that UPPAAL does not accept nested quantifiers. This limitation required some adjustments in following verification queries, analysed in the next chapter. Note that Figure 3.1 outlines that the process of generation ends the automated contribution. Therefore, the verification queries denoting mission properties (both automatically and manually generated) must be verified by the user inside the UPPAAL verifier tool.

Chapter 4

Experiments and results

This chapter shows the results from the proposed methodology, how they were verified and the results obtained from both the generation and verification. It is organised in four sections: one for the experiment settings, containing the general hypothesis for our experiments, the experimental setup and overall results. Next, one for each of the three different experiment scenarios, starting from generation results derived from mapping rules to the verification queries analysed in each case.

Results are from three different RoboMAX [45] mission settings: Two missions from the Food Logistics mission domain (i.e Pickup and Delivery scenarios) and one from the Deliver Goods - Equipment. The food logistics missions share the same HDDL domain file for both missions, but its GM input files are different. The last scenario is a mission about delivering equipment to agents.

4.1 Experiment settings

4.1.1 Experimental setup

The experiments were conducted in UPPAAL in version 4.1.26-1. The code used to generate the NTA for missions was made in Python version 3.10.7, with the use of the uppaalpy library [46, 47] is available at GitHub [48]. Another relevant project is a fork of the original MutRoSe repository [49], modified to output relevant files, as stated in Section 3.2. Additionally, the experiments were conducted on AMD Ryzen 5 4600H with a total of 16GB memory.

4.1.2 General hypothesis

For each of the three missions being analysed, it is intended to display generation results when being compared to the original specification to show that both rules and specification
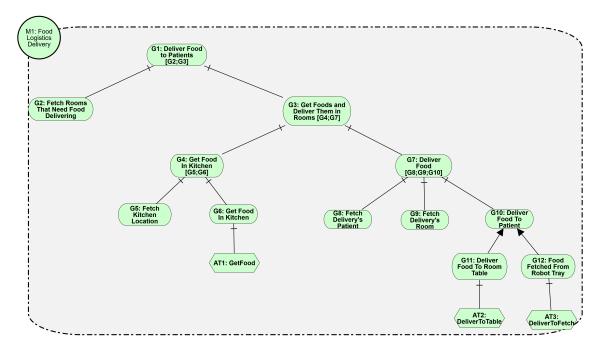
Figure 4.1. Food Logistics - Delivery goal model

adhere to each other. Additionally, we verify properties as queries to validate this approach, properties range from relevant characteristics, deadlock freedom and reachability as defined in Table 3.2.

For RQ1, the hypothesis for this work is that the results yield the same specification from MutRoSe as a NTA by following the mapping rules from 3.4 from MutRoSe and that verification queries are fit for validating the previously stated properties. As for RQ2, the hypothesis is that the verification such as mission correctness and predicates or capabilities affect reachability properties.

4.2 Mission description

4.2.1 Food Logistics - Delivery

Goal Model

The food logistics is a mission used to analyse how robot cooperation can be used to deliver meals to patients who are often unable to pick up a meal tray by themselves. The scenario offers two alternatives to deliver food to those patients: either deliver them directly to the patient, that is, if the patient is able to hold the tray; or deliver to another robot that is capable of delivering the tray next to the patient.

The goal model starts searching for rooms which need delivering in $G2$. Next, the model moves to goal $G3$ which contains two sub-goals: one for the robots to get the meals

in the kitchen ($G4$) and other for delivering them to the patient rooms ($G7$). A sequential annotation in $G3$(i.e. $G4; G7$) already establishes that these tasks cannot be done in no other order. The Figure 4.1 depicts the goal model of the food logistics mission.

During delivery, one important part of the goal model structure is the OR decomposition present in goal $G10$, responsible for defining that either goal $G11$ or $G12$ are executed, but not both. Although runtime operators are primarily associated with changing mission ordering, the OR decomposition plays a fundamental role in this mission to establish which goal and subsequent task will be executed per mission configuration.

Domain definition

As stated before, the domain definition file is used for two separate missions with different goal models. Thus, it contains a lot more method definitions than the ones used in a single mission. The complete file is shown in Listing A.6. In essence, this file domain defines a hospital with patients and robots interacting in methods for various reasons such as object manipulation, delivering and overall logistics inside a health setting.

The abstract tasks used for this mission are as follows: GetFood, DeliverToTable, DeliverToFetch. The GetFood task, as the name suggests, contains the necessary subtasks needed for the robot to get a food meal from a certain location. Then, as the food is obtained, a robot may decide between tasks DeliverToTable and DeliverToFetch, the first one requires no human interaction, but requires the robot to have the capability *manipulation* to be able to deliver the meal correctly. DeliverToFetch needs human interaction, however, it also requires that the predicate *patientcanfetch* is true for the task to be accomplished.

4.2.2 Food Logistics - Pickup

Goal Model

The main goal of this mission is picking up dirty dishes from the rooms where patients are residing in the hospital, in order to achieve that, it must first survey which rooms require pickup of dishes. Next, the main mission is identifying and going through each room to pickup the dirty plates. After dishes have been retrieved, they are delivered to the kitchen.

This GM contains a slightly less complicated task ordering than the last one, where two tasks must be executed in any mission path. This is shown in Figure 4.2, where it is possible to deduct quickly from the CRGM that both tasks must be achieved for a successful execution. This mission contains the remaining methods not used in the last one.

Figure 4.2. Food logistics pickup mission goal model

4.2.3 Deliver Goods - Equipment

Goal Model

This mission scenario from RoboMAX illustrates robots delivering goods or equipment to agents in an uncertain environment. As Figure 4.3 The main goal, of course, is assuring that all the deliveries are made. Differently from the other two previous missions, this one contains fallback operators in 3 goals. In this case, the output will follow rules stated in Section 3.4.

Domain definition

The domain definition file displayed in Listing A.7. Once again, the domain is still a hospital, but storage, agent and obj types were added. Unfortunately, it is noticeable that no predicates are used inside the method definitions, which leaves only action ordering to be generated in the respective templates. This leads to the conclusion that this HDDL file is much more simpler, which shifts the responsibility to the CRGM to deal with variable instances.

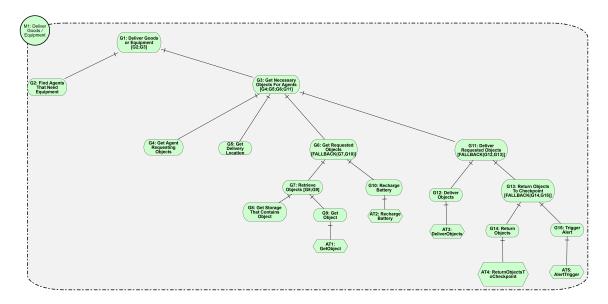Figure 4.3. Goal model for Deliver Goods - Equipment mission

4.3 Results

4.3.1 Profiling results

The generation program [48] took 0.434s for the food logistics mission (in both cases) and 0.433s for the deliver goods. With the cumulative time for the generation process being 0.319s for the food logistics missions and 0.302s to the deliver goods mission. This could be attributed to many generation loops which traverse through the data structures and were not optimised and inner calls made by uppaalpy [46] to other libraries. Base generation performance does not drastically change since most specifications go through the same functions before being properly generated. With the exception of a few additional loops for runtime operators which do not change the general complexity, the overall performance results are rather similar. This could be attributed to the specification and mission sizes which are pretty similar as well. The profiling results were captured using snakeviz [50] and cProfile [51].

4.3.2 Food Logistics - Delivery

On total, 14 templates were generated in UPPAAL, with 6 being directly associated with this execution due to execution paths. The task methods contain many of the original elements present in the original specification. The goal model, at this version, only sustains the original ordering established by runtime and decomposition operators. The goal model template for this specification is displayed in Figure 4.4 and clearly shows that even the OR decomposition was generated correctly, which enables the user to correctly
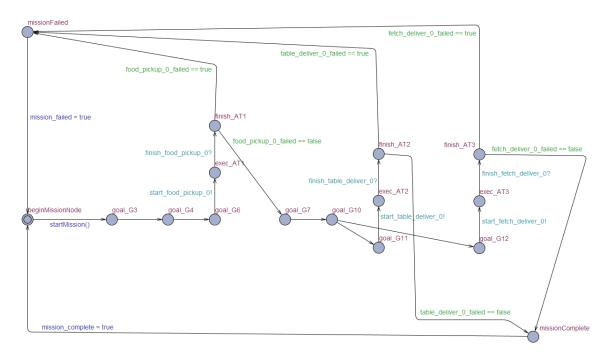
Figure 4.4. Goal model template for food logistics

analyse all mission paths. It is also possible to see that tasks are strictly executed in one of the following orders:

$$AT1 \longrightarrow AT2 \quad \text{or}$$
$$AT1 \longrightarrow AT3$$

(4.1)

Where abstract tasks representation of execution are present in exec_AT and finish_AT locations.

It is also important to discuss the declarations created by this generation, the variables generated are in full conformity with what was expected, even the types for some were derived correctly from specification. As stated before in the generation stage, capabilities are defined in the domain definition without a specific type because the domain definition file does not express directly which robot needs to possess the capability, therefore the addition of a type would imply that the generation knows which robot possess the capability in question, which is incorrect. One benefits in this specification from this fact by not having the necessity to formally assigning another variable to the robot struct every time it is used. This also helps reducing the state space without compromising the specification, since the capability is modelled as a guard constraint in either scenario as shown in Figure 4.5 which corresponds to the template generated for the table-deliver method from the domain file in Listing A.6.

The list of task method templates related to this mission is described below:

1. Food pickup template (temp_food_pickup_0);

45

Figure 4.5. Table deliver template generated in Uppaal

Figure 4.6. Abstract task pattern in FetchMeal inside fetch-deliver method generated for Uppaal

2. Table deliver template (temp_table_deliver_0);

3. Fetch deliver template (temp_fetch_deliver_0);

4. Fetch meal with human template (temp_fetch_meal_with_human_0);

5. Fetch meal with robot template (temp_fetch_meal_with_human_0);

It is important to note that the *DeliverToFetch* contains abstract tasks as subtasks, which will result in a pattern used to trigger these tasks. This pattern is displayed in Figure 4.6. As stated in rule #9 in 3.4, the pattern stands but with its values changed to the actual methods and their respective reference channels and variables.

4.3.3 Food Logistics - Pickup

The same number of templates as the last mission were created. The results are what was expected, however, since our approach is not focused on inferring which robot is respons-

Figure 4.7. Generated template for food logistics pickup mission in UPPAAL

ible for each task, this system declaration requires adjustment of variables to successfully finish the mission. However, it is important that it stays consistent both with GM and domain file definitions. As expected, the generation process also showed to follow the same creation when facing the same patterns rules. One interesting observation from this method is that AT1 possess two available methods: pickup-with-door-opening, pickup-witho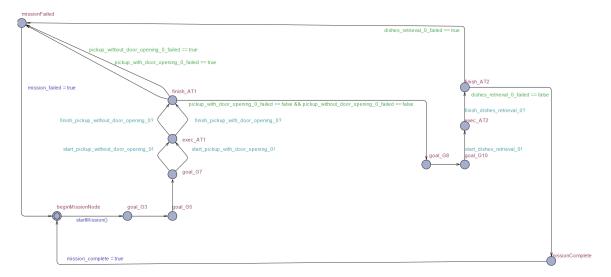ut-door-opening. This is reflected in the generation process as stated in the rule for generation of tasks (i.e. Section 3.4) and its behaviour is present in Figure 4.7. Additionally, it is possible to also see the transition where the task finishes successfully with two guard values in a single boolean clause. This clause was divided in the missionFailed target transitions for this mission to improve readability for the model.

4.3.4 Deliver Goods - Equipment

The results were very positive concerning mission ordering for the goal model template, the method templates, however, fall short due to not having any available predicates that could work as precondition or effect on task level templates. Figure 4.8 shows the UPPAAL goal model level template and depicts how fallback operators are implemented, by comparing with Figure 4.3, it is possible to see that structures follow the CRGM execution pattern, however, it is possible that the execution paths are not clear. In order to illustrate all successful execution paths concerning tasks, please consider the equation below:

Figure 4.8. Generated template for deliver goods - equipment mission in UPPAAL.

$$AT1 \longrightarrow AT3 \text{ or}$$
$$AT1\_fail \longrightarrow AT2 \longrightarrow AT3$$
$$AT1 \longrightarrow AT3\_fail \longrightarrow AT4$$
$$AT1 \longrightarrow AT3\_fail \longrightarrow AT4\_fail \longrightarrow AT5 \tag{4.2}$$
$$AT1\_fail \longrightarrow AT2 \longrightarrow AT3$$
$$AT1\_fail \longrightarrow AT2 \longrightarrow AT3\_fail \longrightarrow AT4$$
$$AT1\_fail \longrightarrow AT2 \longrightarrow AT3\_fail \longrightarrow AT4\_fail \longrightarrow AT5$$

Equation 4.2 depicts possible execution paths where the abstract tasks annotated with the $\_fail$ suffix means that they have failed execution. It is understandable how Figure 4.8 may appear confusing, but the goal model itself offered many alternatives concerning method ordering, hence all the transitions generated.

4.3.5 Properties verification

This section comprises all queries made in UPPAAL (i.e. the expressions made in TCTL syntax for property verification). They are displayed in Tables 4.1, 4.2 and 4.3, one for each mission previously described.

Property ID	Formal description	Expression in UPPAAL	Result	Elapsed time
#1	Deadlock freedom	A[ ] not deadlock	Success	0,002s
#2	Goal G6 is neeeded for mission conclusion: For all paths, if the mission is complete it implies that the goal G6 is also complete (AT1 did not fail)	A[ ] mission_complete imply not food_pickup_0_failed	Success	0,002s
#3	Mission complete: For all paths, if the mission is complete it implies that AT1 and AT2 or AT3 were completed successfully	A[ ] mission_complete imply (not food_pickup_0_failed) and (not fetch_deliver_0_failed or not table_deliver_0_failed)	Success	0,002s
#4	In all paths, if the capability manipulation is not set, then the method fetch_deliver will eventually fail	not manipulation –> fetch_deliver_0_failed	Success	0,002s
#5	Reachability, a mission has a path of success in this given configuration	E<>mission_complete	Success	0,002s
#6	Reachability, a mission has a path of failure in this given configuration	E<>mission_failed	Fail	0,002s

Table 4.1. Properties verification for Food Logistics Delivery mission

Food Logistics - Delivery

The properties verified for the food logistics mission are: reachability (i.e. if a mission is capable of eventually reaching a successfull or a failure state, denoting its end) and deadlock freedom (if the system does not reach a deadlock state). Other relevant characteristics include mission ordering correctness and a capability which influences the mission achievement.

Property #1 represents deadlock freedom, this query format is default and it is generated for all missions, properties #2 through #6 were manually inserted. Property #2 and #3 show mission ordering strictness, which means here that the mission order derived from the specification files still holds in the verification system, in order to show its correctness. For instance, property #2 states that AT1 (i.e. food_pickup method) must be successfully executed to enable the mission to succeed. One can see from Figure 4.1 that this is correct, since G3 fails from not executing AT1. As for mission successful completion, Equation 4.1 shows the available execution paths, which are verified through their methods failure inside property #3. Both queries result in success, pointing out that the generated template for the goal model is in accordance with the specification CRGM. Next, property #4 is a relevant characteristic where the manipulation variable is evaluated in the fetch_deliver task. The query essentially states that, if the capability is not enabled, then the method will invariably fail in the future. The last two properties (i.e. #5 and #6) verify if the mission is always able to reach a failure or success, since UPPAAL verification is based on the present configuration, property #6 fails due to non-existent failure paths in this configuration since both capabilities are true, while, in contrast, property #5 is satisfied.

Property ID	Formal description	Expression in UPPAAL	Results	Elapsed time
#1	Deadlock freedom	A[ ] not deadlock	Success	0s
#2	For all paths, for goal G7 to be satisfied (and goal G8 to be reached), either one of the task AT1 methods (pickup_with_door_opening or pickup_without_door_opening) must be satisfied	A[ ] var_goal_model_template.goal_G8 imply (not pickup_with_door_opening_0_failed or not pickup_without_door_opening_0_failed)	Success	0s
#3	For all paths, mission is completed successfuly if and only if AT1 and AT2 do not fail	A[ ] mission_complete imply (not pickup_with_door_opening_0_failed or not pickup_without_door_opening_0_failed) and not dishes_retrieval_0_failed	Success	0s
#4	Reachability, a mission has a path of success in this given configuration	E<>mission_complete	Success	0s
#5	Reachability, a mission has a path of failure in this given configuration	E<>mission_failed	Fail	0s

Table 4.2. Properties verification for Food Logistics Pickup mission

Food Logistics - Pickup

Food logistics pickup contains only mission ordering correctness as relevant characteristics properties, since no predicates as preconditions or effects are available for this mission. Property #2 stems from the same mission ordering correctness issues explored in last mission. In this case, it is possible to see from 4.2 that goal G7 is satisfied by the execution of AT1 (PickupDishes) which, in turn, possess 2 available task methods: pickup_with_door_opening and pickup_without_door_opening. Thus, it is possible to reach task success by executing one of them. The graph then progresses to the next goal to be executed, which is G8. Therefore, it is safe to conclude that G8 is only reached if G7 is satisfied. Property #3 is used to evaluate mission conclusion successfully from the ordering correctness from the point of view of task execution, which means that it must assure that AT1 and AT2 are executed, hence property #3 states the conditions for that to be achieved. Property #4 and #5 state reachability issues, and their results are coherent with what was expected, since this configuration does not possess failure paths with the current configuration.

Deliver Goods - Equipment

The properties derived from this mission do not come from the domain file because, as stated before, the domain files do not possess any predicates used during the execution of tasks. Property #1 stands for deadlock freedom, properties #2, #3 and #4 investigate mission ordering correctness, finally, #5 and #6 are reachability properties.

In property #2, it is important to note that since there is a fallback runtime operator (see Figure 4.3), this means that the task may be completed successfully in two separate conditions:

Property ID	Formal description	Expression in UPPAAL	Results	Elapsed time
#1	Deadlock freedom	A[ ] not deadlock	Success	0s
#2	Meant to fail, this one shows that G11 is executed if AT1 or AT2 (if AT1 fails) successfully execute, not just AT1, once again showing that the fallback structure is sound	A[ ] var_goal_model_template.goal_G11 imply not object_get_0_failed	Fail	0s
#3	For all paths: Goal G11 is reached if AT1 or AT2 (in case AT1 fails)are successfully completed, it also means that G9 is completed	A[ ] var_goal_model_template.goal_G11 imply not object_get_0_failed or object_get_0_failed and not battery_recharge_0_failed	Success	0s
#4	Mission complete is achieved by adopting one of the execution paths available in Equation 4.2	A[ ] mission_complete imply (not object_get_0_failed) or (object_get_0_failed and not battery_recharge_0_failed) and (not objects_delivery_0_failed) or (objects_delivery_0_failed and not object_returning_0_failed) or (object_returning_0_failed and not alert_trigger_0_failed)	Success	0s
#5	Reachability, a mission has a path of success in this given configuration	E<>mission_complete	Success	0s
#6	Reachability, a mission has a path of failure in this given configuration	E<>mission_failed	Success	0s

Table 4.3. Properties verification for Deliver Goods - Equipment mission

1. AT1 (with method object_get) finishes successfully;

2. AT1 fails and AT2 (with method battery_recharge) finishes successfully.

Given those two conditions, it is clear why property #2 fails, since it does not take option 2 into account. As opposed to property #2, property #3 takes the alternative path into account, thus it is satisfied since both paths are the only available paths to reach goal G11 and satisfy goal G9.

Property #4 pushes the structure of nested fallbacks even further: as the last line in Equation 4.2 the verification query evaluates the execution path where AT1, AT3 and AT4 fail, but the necessary tasks are executed and the mission completes with success. Reachability properties #5 and #6 are the same as the other missions, however, #6 yields a success result, this is because the failure states are automatically added when a task is inside a fallback runtime operator, in order to stay true to the specification, where the task might fail, but also to analyse the execution paths spanned from failed tasks. Thus, since failure is by default an option, both reachability properties are true at the same configuration.

4.4 Complexity issues

Given that all locations are currently derived from HDDL and the GM, an analysis of the generation process itself is necessary to evaluate its space and time complexity. In the sections below, we will analyse the complexity in the generation stages for each input file, which will help the reader to assess not only the complexity of this approach, but also its current limitations for future scalability.
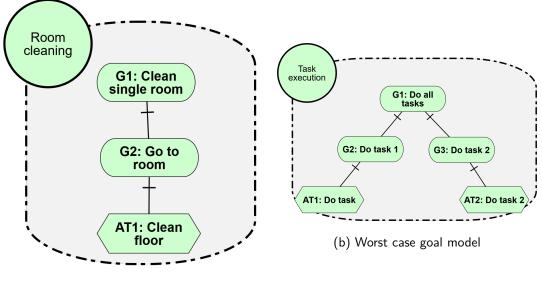
4.4.1 HDDL

When considering hierarchical domains in UPPAAL, the generation process does not discriminate between what methods are used or not. Therefore, all methods present in the domain file are generated and instantiated as UPPAAL processes, even if they do not take part in a specific mission. Recall that, all methods become one UPPAAL template, and all subtasks are individually created. This applies even if the same action is used for two different methods or in two different subtasks in the same method. Additionally, the branching paths preventing deadlocks in templates add a constant number of new locations, also increasing the state space. Thus, consider that for s subtasks present in all methods that the same amount of locations will be derived from the generation process, additionally, we have a constant number of additional locations which bear a k constant number for all subtasks with preconditions or capabilities needed. Which leads to time complexity $\mathcal{O}(s + k) = \mathcal{O}(s)$. Therefore, we conclude that the generation takes linear time. As for the space complexity, the generation itself also needs to use data structures containing all methods with all subtasks, thus the complexity also stands at $\mathcal{O}(s)$.

4.4.2 GM

The CRGM generation in UPPAAL uses a structured tree for generation, where a top-down creation for the related tasks and goals takes place. Before starting the generation process, as stated by rule #12 in Section 3.4, the removal of some unnecessary goals (i.e. goals that are not included in the current approach such as query goals or achieve goals) takes place, where they are discarded from the generation process. After that, the node from the tree which contains all tasks as children is the starting node from the generation, which then generates all nodes as locations (and uses a particular generation pattern, should the node contain a runtime operator) until a task or a goal leaf is reached. In case of the latter, the goal leaf is also discarded since it contains no tasks and therefore does not hold any importance for the current version of this project. Thus, it is possible to describe the generation process in terms of time complexity could be expressed by the following equation:

$$\mathcal{O}(n - u_g - l_g) \tag{4.3}$$

Where n is the total number of nodes in the goal model, where the tree generation time in UPPAAL is $\mathcal{O}(n)$, subtracted by the u_g unnecessary goals and l_g leaf goals with no tasks. As for state space, the tree structure is broken down as lists of lists, resulting in a space complexity of $\mathcal{O}(n)$ for the generation process.

(a) Best case goal model

(b) Worst case goal model

Figure 4.9. Best and worst case scenarios for generation of the goal model

Worst case and best case scenarios

For the best case, it is easy to immediately deduct that the smallest GM (i.e. one with the minimal valid amount of nodes) is the best case, because the generation would take less time. Given this scenario, consider a valid MutRoSe CRGM depicted in 4.9a with exactly three nodes: two goals, a root goal, a goal preceding a task and the task itself. No goals would be excluded, so the generation process is clearly $\mathcal{O}(3) = \mathcal{O}(C)$. This is the best case as the generation would not be void, which would compromise the generation as the methods generated in task templates depend on channel triggering by the GM template.

In contrast, the worst case scenario is one which:

- No unnecessary goals or leaf goals (i.e. u_g and l_g) are present;

- All nodes in the GM are generated.

With that in mind, the figure depicted in 4.9b meet this criteria, with the generation complexity generating all 5 nodes (i.e. $\mathcal{O}(5)$). Now extrapolate this example for a goal model with n total nodes, it is easy to see that the generation would take $\mathcal{O}(n)$ time. Thus, the worst case scenario for GM generation takes linear time ($\mathcal{O}(n)$.

4.5 Discussion

To test the generation tool, seven different scenarios were executed in this approach, all of them generated outputs for UPPAAL. However, two of them gave inconsistent gen-

HDDL	Generated in NTA?
Types and variables	Yes
Predicates	Yes
Tasks	Yes
Methods	Yes
Ordered subtasks	Yes
Subtasks	No
Actions	Yes
Capabilities	Yes
Goal Model	
OCL Statements (monitors/controls syntax)	No
Task attributes	No
Goal Types	Perform goals only
Divisible and group attributes	No
Runtime Operators	Partially (parallel not implemented)
Mission ordering	Yes

Table 4.4. Summary of MutRoSe elements generated to UPPAAL

eration results and one contained syntax errors, this can be attributed to unexpected nested runtime operations, which have many edge cases able to compromise the generation process. The three mission scenarios described here were carefully analysed and no generation errors concerning specification flaws were found.

Generating mission ordering as a NTA took more time than expected during the project due to many edge cases, this also hindered progress in generating other MutRoSe GM structures, which were critical to full mission verification. The results generated for the domain files were very interesting as many of the domain structures are correctly generated as a verifiable model, with the exception of non-ordered subtasks, which were not explored in missions analysed. Table 4.4 brings a summary which lists the elements from MutRoSe that were generated or not in the UPPAAL NTA, future works aim to contemplate more CRGM elements in a near future.

4.5.1   Scalability issues

Scalability is an issue which needs to be addressed for future iterations of the project. However, it is suggested that the generation would not differ in overall complexity as only new runtime annotations and divisible and group attributes would add constant time complexity for template generation. This due to the fact that the levels described in this approach would be further implemented with more variables and locations without

the need for more templates. As for UPPAAL, it is still uncertain that these additions would impact severely on the state space, new experiments must take place to assess the robustness of the tool in larger state spaces. Another alternative is switching to statistical model checking for the verification of properties if the current approach is not possible.

In conclusion, the solution is incomplete for MutRoSe specifications due to its lack of many CRGM elements, but holds interesting study points as to where it is possible to evolve and integrate this solution. One other interesting remark is expanding MutRoSe specification such as timed constraint to verify more properties with UPPAAL. The answers for RQ1 and RQ2 are as follows:

RQ1 (How to automatically verify mission specifications of heterogeneous MRS from a high-level perspective?) Through the automatic generation of a program [48], it was possible to not only generate verifiable models for mission specifications from MutRoSe but also to verify missions from a high-level perspective (predicates, goals, capabilities, abstract tasks, etc.). Although not all elements were mapped in this generation process, it has been shown that this the automatic verification is definitely feasible.

RQ2 (Is it possible to extract relevant characteristics from MRS mission specification models as verifiable properties?) Some properties such as capability were submitted through verification and have shown interesting results regarding method or mission reachability. Additionally, the automatic generation of a verifiable model from a MRS mission specification alongside the manually inserted properties verified have shown that mission specifications as verifiable models might provide more insight concerning mission properties to designers.

4.6 Threats to validity

Although the UPPAAL models were generated automatically, many edge cases in the program implementation could greatly increase the experimenter bias. Additionally, repetitive testing was made during the creation of the generation process, thus compromising internal validity to an extent. On the other hand, these claims could be countered as many generation stages are the same for the three experiments analysed in this work. Additionally, mapping rules have not changed during the elaboration of the methodology through experimental stages, increasing confidence in internal validity. Additionally, since it is a generation program, the results are not affected by time features.

Since MutRoSe is still a DSL, the generation process is obviously tethered to its syntax, which means that the generalisation of the experiment is not applicable to other

frameworks or other MRS without adjustments. Additionally, only functional samples were analysed, thus, external threats may include selection bias and sample features. A factor that improve the external validity is that the generator of UPPAAL is a project freely available [48] for replication.

Chapter 5

Related works

The MRS area contains many works ranging from operation and planning to specification and verification. Regarding verification, a survey made in 2019 by Luckcuck et al. [16] gathered 25 works using model checking in the literature in various tools. The most used tool was SPIN [52], containing 5 works, followed closely by others such as PRISM [53], UPPAAL [5] and others. Table 4 in the same survey indicates that model checking the most popular formal approach for verification in MRSs, containing a total of 32 works, more than all other formal approaches surveyed combined (24). Thus, this once again shows that, while there's not a predominant tool used in model checking since they all possess many different characteristics when it comes to implementation and could be applied to many different domains, model checking is often the most adopted formal verification method as it provides an outlined mathematical proof as to why properties would hold or not in the states specified.

The MRS field contains many papers using formal verification, since safety, liveness and reachability are common properties evaluated during design or execution time in such systems. Additionally, as mentioned before, there are various formalisms and tools to tackle verification problems on such systems, but not many of them contain a pipeline of specification and validation inside the same framework. Therefore, works containing a integrated framework for verifiable MRS will be outlined in this section. Additionally, although some approaches accommodate other types of systems, such as Self-Adaptive Systems (SAS), the works will be focused in single or multiple robotic applications (namely MRS). Another important aspect of MutRoSe is that missions are described on a high-level, meaning that no specific mission context other than being accomplished by MRSs must be considered before designing a mission inside the framework. One of the comparison levels with other works is if they offer a high-level, top-down approach. Some other concerns such as heterogeneity, which formalisms and tools are used for verification in each work (e.g PCTL, TCTL, Process algebras, etc.), lastly, which properties are being

verified, such as safety, reachability, security, dependability, etc.

5.1   Translating RoboSim models to UPPAAL

The work of [35] aims to translate RoboSim [54] models as UPPAAL Network of Timed Automata (NTA)s automatically following a strict set of timed automata patterns and rules, it is also restricted to a specific context of RoboSim metamodel mission i.e the plugin only generates NTAs for RoboSim models. The work presented in this paper can describe any mission setting described inside the MutRoSe framework, this indicates that there is a trend in establishing a streamlined and automatic verification process from the very DSLs specification in recent works. Additionally, the work also provides a combination of the NTA and the Network of Stochastic and Hybrid Automata (NSHA) to enable verification of Weigthed Metric Temporal Logic (WMTL) properties using UPPAAL with the Statistical model checking (SMC) extension (namely UPPAAL SMC). MutRoSe specification does not contain any weighted properties, likewise, the models do not contain this. Another interesting point of comparison between the two works is what are the input languages for the generated model. RoboSim models are diagrams similar to Unified Modeling Language (UML) notation, containing roughly two main module levels: a syntactic unit module, which models the robotic system by specifying the interfaces of the robotic platform, and the software controllers module, which contains the behaviour of controllers running in parallel with the unit module.

While MutRoSe focuses on mission specification and decomposition of high-level mission plans for MRS, RoboSim is a framework used for modelling robotic simulations in diagrams using state machines using a timed syntax. The work in [35] transforms models designed in RoboSim to UPPAAL. Both works converge in translating a specification language to a verifiable model using UPPAAL as the verification tool. Unlike the latter, it is much more difficult to generate a automata from a set of tasks. Since RoboSim models describe a cyclic simulation, Zhang et al. work provide a low-level abstraction transformation process, whereas this approach generates abstractions for verifying abstract plans, where no detailed explanation of how the tasks will be carried out is provided, only their ordering (if not partial). One shared similarity is that the two approaches generate default verifiable properties such as deadlock-freedom from their translation. The case study is a RoboSim model using the Alpha Algorithm, a algorithm studied in other MRS works [55] that aims to aggregate a swarm of robots. The work for this paper focuses on high-level specification in order to be able to accommodate more mission contexts.

5.2 The Esterel framework

The work of Kim et al. [56] uses the Esterel framework, which consists of a programming language, a graphical simulator and the XEVE model checker [57], to formally verify a robot home assistant named Samsung Home Robot (SHR) developed by Samsung Advanced Institute of Technology (SAIT). XEVE uses model checking on FSM models generated by the programming language after compilation, however temporal logics are not used in verification, instead, it uses compositional techniques to reduce the state space complexity with the removal of redundant states in a process similar as OBDDs in MCMAS. and then implement verification using observers: reactive components placed in parallel with the main program which are triggered by signals of success or failure during simulation stage, where all possible inputs are tested to cover all possible combinations. It is possible to check for safety and liveness properties within the system by using synchronous observers as outputs.

The implementation of robot systems is considerably low-level, signals are used for interaction between components, including timing intervals and counters. Thus the user can describe each behaviour of the robot accurately, which incur in many code lines. However, the very use of signals inside the modules allow for the user to preemptively creating events for error handling and guarantee safety in SHR as well as other properties. While this allows for more control from the designer when defining properties, it is necessary that the events are properly rigged to guarantee safety, as such, requirements must be thoroughly inspected before they are specified in Esterel. Another advantage of specifying interactions when it comes to debugging is that since signals are the only communication channels between modules, then the testing among components is simplified by only observing their signal states.

The Esterel framework and the model checking approach for MutRoSe differ in many levels, ranging from the verification techniques used to the very scope of robotic applications (i.e. low-level behaviour of a single robot *versus* high-level mission goals and tasks). Some similarities both works share is that both MutRoSe specifications and the SHR are designed inside their respective frameworks and have safety properties assessed through model checking. Another coincidence between both frameworks is the use of graphical simulators to assist designers in the specification process, likewise, the representation using FSMs shares some similarities with NTAs used UPPAAL.

5.3   The BIP framework

There is a clear difference between top-down and bottom-up approaches: as mentioned before, although bottom-up approaches are more likely to capture all aspects of a certain background as they are more specific, top-down ones tend to create a more simple abstraction for MRSs. This is useful for designers who do not have a lot of time to design a MRS from scratch, however, it is important to note that hurrying modelling stages may conceal a lot of design-specific problems. Many MRSs use a bottom-up approach to ensure verification, as shown in the case of [58].

The paper adds the idea of a component-driven design in robotic systems combining the strengths of $G^{en}oM$ and the (Behavior, Interaction, Priority) (BIP) Framework [59]. Furthermore, the combination of both tools enable automatic generation of correct robotic software in $G^{en}oM$ through specified behaviours in BIP for the $G^{en}oM$ environment. Components are described using transition systems written in C/C++ through BIP. By describing components in an individual level, it is possible to design MRSs in a incremental and safe manner due to modelling and verification being done on the component itself and then gradually on others as design progresses, reducing the cost of correction by detecting the problem at an early stage. This methodology is applied to a autonomous rover robot as the case study where constraints defined by the BIP framework hold in generated code. The verification used inside the toolset is a model checker tool called DFinder [60] which also uses a compositional verification method. With the use of component and interaction invariants (i.e. descriptors of atomic constraints and constraints between modules, respectively, described as invariants), it is possible to check if properties remain satisfied incrementally during verification. The input program for verification is written in the BIP language. The properties verified are: deadlock-freedom and safety.

While this is not a MRS, this work can be easily adapted to a multiple number of robots in the same incremental manner. This study also shows an interesting verification technique: by using the negation of a defined property and checking if there is not a state where the negation occurs, we can safely assume the property holds. Otherwise, the potential violation can be investigated further as the state is shown by the verification tool. In UPPAAL, when a property described in TCTL fails to hold, it is possible to check which state, i.e. location, is responsible for the violation. This idea is used in this work to show locations of the negated property when there are more locations of the property itself during verification queries. Other similarity with the component-based construction is that control mechanisms coordinate connection and functional elements by providing a control flow, enforcing a clear separation of responsibilities, likewise, in the verification stage of MutRoSe, aside from developing a different abstraction for each file, the generation of each task must be done in a hierarchical approach to assure a harmonic

interaction with the upper levels.

5.4 MissionLab and VIPARS

The work of [4] offers autonomous verification of behavior-based controllers created in the Configuration Network Language (CNL), a component of the MissionLab Mission Specification System [61] by translating specifications to Process Algebra for Robot Schemas (PARS) and then submitting them to processing with the verification module Verification in Process Algebra for Robot Schemas (VIPARS). Additionally, the paper provides a feedback loop by returning the verification results to the human operator. Figure 5.1 illustrates the overview of the architecture. It is possible to notice that the performance criteria are distinct from the models and can be parameterised in order to find the best tuning for the robot controller. Specification of the models is done in CNL, which is a superset of C++ designed to express the separation between behaviour implementation and its integration with other previously defined behaviours, thus, a top-down construction of robotic applications. PARS is the specification language for formal verification. It is specialised in verification of concurrent systems, as such, it is capable of representing robot controllers and their hardware as well as the environment and how it affects the MRS through interactions. In process algebras, the composition of processes follows a stop and an abort condition, in a way that basic processes can be connected with others and generate complex behaviour patterns. The automatic translation of controllers into PARS is done following a strict set of rules of lexical grammar parsing using Flex and Bison [62], two parser tools widely known that have also been used for the work of MutRoSe during its parsing of HDDL. As a downside, the paper reports that it is more suitable to design and select a set low-level behaviours rather than designing intricate and complex behaviours for the translation to PARS primitives. This could be attributed to the corresponding number of CNL nodes generated and a state-space explosion problem during translation as well as accuracy when parsing behaviours to PARS.

Although the verification using VIPARS uses process algebras, both process converge in using a similar input, the CNL language converts its controllers into a FSA that is translated as a PARS, while the base format for UPPAAL specifications is a NTA which is closely related to the other automata notation. Another point of convergence is that the translation process starts from a high-level mission structure. As said before, the complex processes found in the upper levels of PARS are compositions of more primitive nodes, this is a strong correlation where the mission process specification start its description from the higher levels of behaviour and starts specialising behaviours as needed. However, both works differ when analysing description of process algebras and temporal
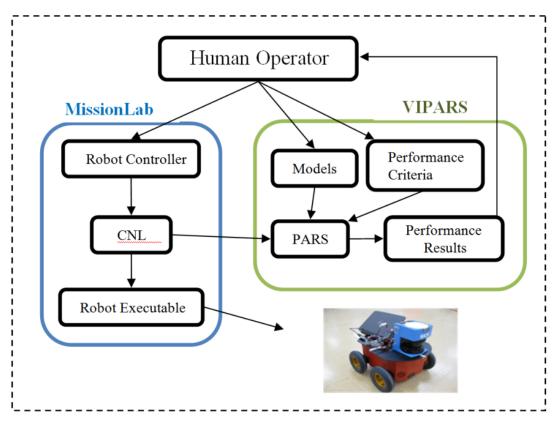
Figure 5.1. Overview of the architecture used in [4]

logics, while it is necessary to describe interactive environment behaviours inside VIPARS to accurately represent the MissionLab specification, which is a robot search mission for biological weapons in an unknown environment, one does not need to outline contextual specifications depending on the mission context for MutRoSe, although it is possible with the use of contextual annotations. As such, the performance criteria defined further distinguish both framework goals when it comes to the properties verified: the VIPARS framework is able to define timed criteria for its missions. For instance, in the search mission, a performance criteria chosen was that the robot must find the target within 60 seconds. In conclusion, since VIPARS framework is concerned with specific performance criteria i.e qualitative properties, the verification processes between both works are fundamentally different when comparing objectives, nonetheless, it is possible to compare both works when assessing robot specification and automatic translation to verification models.

5.5 vTSL

verifiable Task Specification Language (vTSL) [63] is a DSL used to specify task trees which allows formal verification of a set of previously defined safety and integrity restric-

tions of a robotic application using the model checker SPIN [52]. It is important to note that it is also possible to define task constraints to be checked against within the language. Additionally, the paper offers a automatic transformation of vTSL specification models into Promela models, i.e. the input language for SPIN, and demonstrate its usability and scalability through experiments that directly use the behaviour specification models. The example application used for this work was a logistics use case. vTSL uses only a textual language similar to C/C++ to describe the behaviours of a robotic system, one limitation of this approach is only being able to describe one robot behaviour in favor of more elaborate specification when compared to other works [64]. The specification language has a fundamental block called *action*, an action may trigger other single or concurrent actions and return type values, additionally, each can contain input parameters, thus enabling to relate actions with one another. Inside a task it is possible to define behaviour blocks, responsible for defining which action will be taken based on the current conditions. Since actions are the basic building block, it is safe to conclude that the designer is likely to follow a bottom-up approach when defining a task tree. It is important to note that vTSL also has interface connectivity with Robot Operating System (ROS) [65] ranging from messages to services.

During translation, each action and component stub in vTSL is created as a Promela component named *proctype*, which is a similar structure to actions in the specification language. As Promela also shares a resembling C structure, many of the transformations for the verification model are considerably straightforward. After generating assertions and all execution paths in Promela, the model is then submitted for verification inside SPIN. The verification checks for deadlock-freedom and if assertions are satisfied in all execution paths. As mentioned before, SPIN uses LTL as formalism for model checking, assertions are directly generated from the specification model as LTL properties. However, more complex behaviours must be manually specified in the model.

Compared to the verification approach proposed for MutRoSe, this work focus on single robot behaviour and is mission-oriented, being able to define in a reasonably low-level what the specific robot actions are. Conversely, the verification for MutRoSe is not concerned on which behaviours are allowed during runtime, rather, it focuses on which actions should be taken, but does not go into detail on how they will be carried out, only their sequencing. Another interesting comparison is the difficulty of the translation, whereas there are graphical inputs in the CRGM for MutRoSe, vTSL is a text-only language similar to C, same as Promela. UPPAAL also shares C/C++ notations, but has a graphical notation as well. Therefore, it is safe to assume that the properties are preserved with more precision from the specification model since it has a more direct translation process. Moreover, the bottom-up approach for modelling vTSL differs from

the verification done in MutRoSe, which is high-level and top-down. Finally, one great point of difference is the depth of each abstraction, as vTSL is focused on one robot, it is much more likely to go into further details to explain the abstractions and behaviours it must follow, while MutRoSe is more suitable for a mission specification rather than a robot specification.

5.6 Translation of high-level models to SMV

In National Aeronautics and Space Administration (NASA) robotic applications, it is imperative that autonomy of systems is concise, due to long-term mission applications or environments where human involvement would be too dangerous or too expensive. The work of Pecheur and Simmons [66] presents formal verification on Livingstone, a model-based health monitoring system developed at NASA using Model-Based Processing Language (MPL). The approach used was to design an automatic translator of mission specifications to the SMV model-checking language, next, the generated models were checked using the SMV model checker and the results were returned to the source language in order to assist the designer with the diagnosis process. Additionally, the translation was done to Task Description Language (TDL) task descriptions of mobile robot systems. The properties in SMV are expressed in CTL.

The translation process benefits from the fact that MPL and SMV specifications are reasonably similar, for instance, the synchronous concurrency semantics are present in both languages. One downside was hierarchically mapping variables to SMV: while the MPL models do not possess such syntax, SMV variables are linked to modules. Therefore, one incorrect mapping could compromise the entire hierarchical chain in a specification. The solution was to map the variables in all three parts of the translation and select them accordingly. Top-level modules are defined using a similar syntax in MPL, where some specifications for verification are already in CTL, making the translation to SMV only a matter of syntax. Specification patterns are used for common properties such as reachability. For other properties such as completeness and consistency, some disjoint nodes are used to prevent transition synchronisation issues during SMV translation. Thus, this enables for two properties from the same node to be able to hold in a specification pattern. Not many details are given about the robot application, but it is possible to see a simplified TDL code excerpt dealing with the deployment strategy specification of a MRS in a related work by the same authors [67].

High-level models often hinder some of the more specific implementations to the user, this is also true for the work of Simmons and Pecheur: the specification modules only present how actions will be synchronised, and if there are any concurrency issues where

properties would not hold. Another similarity with the verification done for the MutRoSe is the automatic translation to a specification model. Due to language syntax, the process seems to be more straightforward than the one done in MutRoSe, as a textual and a graphical language are being translated to a NTA. Another interesting point is the use of the translation process in more than one language (i.e. MPL and TDL), the same could be said for CRGM and HDDL as they have different mapping rules inside the generation process, however, both outputs do not relate in the work of Simmons and Pecheur, as they are discussing different autonomous applications. Finally, another interesting point is that TDL is similar to HTN, where tasks are described as hierarchical, showing that HDDL and TDL indeed share some similarities and both are used to define multi-robot applications.

5.7   Related works comparison

A comparison between the related works and the current work proposed is done in this section. The comparison Table 5.1 outlines the differences between all works, in the list below are the axis of comparison explained and why they are relevant for MRS.

- High-Level Specification: A high-level specification works the same as a high-level abstraction: it hinders some layers of the implementation in order to prioritise other more important concerns. One major concern of the Verification and Verification (V&V) field in robotics is delimiting the scope of verification techniques, since many robot applications have completely different objectives when it comes to their specific settings. For example, in service robotics it is difficult to assess which properties need to be covered to assure correct functioning in a HRI scenario as there is not a single technique that would provide complete coverage [68]. Therefore, defining whether it is a high-level specification or not is crucial when defining which properties are verifiable.

- Tools and formalisms: Verifiable frameworks can use one or more formalisms. Since this work deals specifically with verification of robot specifications, it is very important to know and compare what other formalisms are being used in other frameworks in order to evaluate the current work impact on the current state-of-the-art. Another factor that could impact the verification is the tool adopted: UPPAAL, for instance, does not accept nested operators.

- Design-time verification: In verification, there are many techniques to achieve properties verification, some of them can offer complete coverage such as model checking,

	Feature					
Work	High-Level?	Verification Formalism and Tool used	Design-Time?	Top-Down (TD) or Bottom-Up (BU)	Automated?	Mission-Oriented
[35]	✓	Model Checking / UPPAAL	✓	TD	✓	✓
[58]	✓	Model Checking / DFinder	✓	BU	✓	✗
[56]	✗	Model checking / Xeve	✗	BU	✗	✗
[4]	✗	Process Algebras /VIPARS	✗	BU	✓	✓
[63]	✓	Model Checking / SPIN	✓	BU	✓	✗
[67, 66]	✓	Model Checking / SMV	✓	TD	✓	✗
This work	✓	Model Checking / UPPAAL	✓	TD	✓	✓

Table 5.1. Comparison chart of related works

since it evaluates all possible combinations. However, there are other types of verification, such as running simulations and establishing a level of confidence based on the test results for a given number of experiments.

- Top-down or bottom-up construction of MRS: An application built from the most fundamental fragment is more likely to present less errors since it could be verified from the very beginning. Whereas top-down approaches would benefit from having less details to worry during first design iterations. Both approaches have their advantages and flaws, however, it is important to notice that, similar to high-level and low-level specifications, this construction design also impacts on the types of properties verified, moreover, how they are specified by each work.

- Automated or manual verification: Manual properties are more likely to capture specific properties of a MRS. The disadvantage of manually defining them is causing an overhead during system design for extra translation to a specification language, often undesirable for many stakeholders while also leaving them prone to errors [67]. Automatic verification of properties emerges as a interesting alternative to check such systems without having to worry with writing in a verification language. While this saves time, many related works studied here may need additional coding to verify non-common properties [66, 63].

- Mission-oriented: As the verification for MutRoSe is mission-oriented, it may differ from other verifications, which can be more focused on the overall robot behaviour, excluding mission concerns. This is valuable for comparison since some properties would not be feasible if they are not mission-oriented. For instance, the work in [4] evaluates if the mission can be completed under a certain time, while Heinzemann and Lange [63] have shown the scalability performance of their approach. Mission properties can only be evaluated if mission aspects are being placed in specification.

As shown in Table 5.1, the work of [35] possess very similar characteristics to the ones being evaluated in this work, however, it is not high-level as movement constraints

and other layout specifics are taken into account inside the RoboSim model, whereas in this approach they are not. The work in [58] is also not high-level due to its incremental (bottom-up) construction of individual components leading to a connected system of components, which takes communication and other low-level aspects into consideration. Additionally, the work is heavily focused on component specification and not mission-oriented, while this approach is not concerned with low-level implementation of communication modules nor how components behave individually and collectively outside a mission context. The work of [56] is clearly low level, as communications signals between components are included in the robot design, the automated verification process proposed here is only palpable for high-level mission specification, thus properties involving low-level signal communication would not be in the scope of verification. Other diversion point is that the approach using the XEVE model checker is for only one robot, whereas this approach is focused in multi-robot settings.

[4] proposes an automated framework for verification of behaviour based controllers, the formalism uses process algebras and timed constraints in its properties. Although this work uses UPPAAL, such timed properties were not included because there are no timed constraints inside MutRoSe specification, another difference is that UPPAAL uses only temporal logics as verification formalism. Since controllers are made separately and then connected, it is safe to conclude that the approach is also bottom-up, another point of divergence with the work presented in this document. The work in [63] once again brings a single-robot system in contrast with the ones being verified in this work. The second and last difference with the axis of comparison used is that the bottom-up approach completely differs from the top-down verification process used in this work. Lastly, the work of [67, 66] also proposes an automatic generation, one difference between both works is that the high-level specification differs from the ones approached here, where the specification is concerned with the specifics of an inference engine inside Livingstone. This also renders Livingstone verification more component-based rather than mission-oriented.

Chapter 6

Conclusion and Future Work

This chapter contains the conclusions made from this work, along with future works and other important remarks.

6.1 Conclusion

The automatic generation process has proven to be resourceful, however, not all elements from MutRoSe could be added to the generation process. Still, through properties verified, it has already been established that MutRoSe mission configurations may only output success or failure, which is very useful to establish if a mission without fallback tasks may contain a defective execution path which needs correction. By adding the remaining CRGM elements to the Uppaal model, deducting successful execution paths simply by looking at specifications might be challenging.

Verification & Validation has been emerging as one of the most important areas in multiple fields due to growing complexity of systems. MRSs are no different: testing a robot application before deployment is a common industry standard to prevent accidents or flaws during operation. Verification through model checking is a great way to reduce design flaws by thoroughly and exhaustively analysis of multiple execution paths. Model checking is as flexible as it is useful: by comprising a small set of rules needed to specify and verify a system with a few steps, it has been proven during the course of this project why it is the most used technique to verify MRS. MutRoSe may have many more properties left to explore, however, the ones done in this work already show how many of the future works are feasible with formal verification. Therefore, model checking, automated frameworks and DSLs will surely remain relevant for as long as robots evolve their capacities and break more boundaries.

6.2   Future works

Future works include improving the state space of the generated NTAs by reducing control structures which replicate the mission ordering, this is possible by adding committed annotations to Uppaal locations as they are being generated. Other works include expanding the verification of other properties such as CRGM elements left out of this iteration of the project. It is also possible to expand MutRoSe specification to cope with timed constraints and make use of Uppaal clock variables to verify even more properties from a single specification. Other possible addition would be adding knowledge from the world knowledge file, intentionally left out of this approach in order to improve flexibility. However, during the course of this project, it has been identified that it is possible to derive information from the world knowledge without directly using them. Therefore, future iterations of the project might include partial knowledge about the world state without actual instances.

Referências

[1] Liggesmeyer, Peter, Martin Rothfelder, Michael Rettelbach, and Thomas Ackermann: *Qualitätssicherung software-basierter technischer systeme–problembereiche und lösungsansätze.* Informatik-Spektrum, 21(5):249–258, 1998. ix, 6

[2] Ali, Raian, Fabiano Dalpiaz, and Paolo Giorgini: *A goal-based framework for contextual requirements modeling and analysis.* Requirements Engineering, 15(4):439–458, 2010. ix, 11, 12

[3] Gil, Eric: *Mutrose: A goal-oriented framework for mission specification and decomposition of multi-robot systems.* Master's thesis, UnB, 2021. ix, 3, 13, 14, 20, 26

[4] O'Brien, Matthew, Ronald C Arkin, Dagan Harrington, Damian Lyons, and Shu Jiang: *Automatic verification of autonomous robot missions.* In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 462–473. Springer, 2014. ix, 61, 62, 66, 67

[5] Behrmann, Gerd, Alexandre David, and Kim G Larsen: *A tutorial on uppaal.* Formal methods for the design of real-time systems, pages 200–236, 2004. x, 4, 13, 14, 15, 19, 57

[6] Guiochet, Jérémie, Mathilde Machin, and Hélène Waeselynck: *Safety-critical advanced robots: A survey.* Robotics and Autonomous Systems, 94:43–52, 2017, ISSN 0921-8890. https://www.sciencedirect.com/science/article/pii/S0921889016300768. 1

[7] Zhao, Xingyu, Valentin Robu, David Flynn, Fateme Dinmohammadi, Michael Fisher, and Matt Webster: *Probabilistic model checking of robots deployed in extreme environments.* Proceedings of the AAAI Conference on Artificial Intelligence, 33(01):8066–8074, Jul. 2019. https://ojs.aaai.org/index.php/AAAI/article/view/4809. 1

[8] Ahn, Ho Seok, Min Ho Lee, and Bruce A. MacDonald: *Healthcare robot systems for a hospital environment: Carebot and receptionbot.* In *2015 24th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pages 571–576, 2015. 1

[9] Belpaeme, Tony, James Kennedy, Aditi Ramachandran, Brian Scassellati, and Fumihide Tanaka: *Social robots for education: A review.* Science robotics, 3(21), 2018. 1

[10] Helms, Evert, Rolf Dieter Schraft, and M Hagele: *rob@ work: Robot assistant in industrial environments*. In *Proceedings. 11th IEEE International Workshop on Robot and Human Interactive Communication*, pages 399–404. IEEE, 2002. 1

[11] Bi, Z.M., Chaomin Luo, Zhonghua Miao, Bing Zhang, W.J. Zhang, and Lihui Wang: *Safety assurance mechanisms of collaborative robotic systems in manufacturing*. Robotics and Computer-Integrated Manufacturing, 67:102022, 2021, ISSN 0736-5845. `https://www.sciencedirect.com/science/article/pii/S0736584520302337`. 1

[12] Baier, Christel and Joost Pieter Katoen: *Principles Of Model Checking*, volume 950. MIT Press, 2008, ISBN 9780262026499. `http://mitpress.mit.edu/books/principles-model-checking`. 1, 4, 13

[13] Miyazawa, Alvaro, Pedro Ribeiro, Wei Li, Ana Cavalcanti, and Jon Timmis: *Automatic property checking of robotic applications*. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3869–3876, 2017. 1, 5

[14] Konur, Savas, Clare Dixon, and Michael Fisher: *Analysing robot swarm behaviour via probabilistic model checking*. Robotics and Autonomous Systems, 60(2):199–213, 2012, ISSN 0921-8890. `https://www.sciencedirect.com/science/article/pii/S0921889011001916`. 1

[15] Lamine, Khaled Ben and Froduald Kabanza: *Reasoning about robot actions: A model checking approach*. In Beetz, Michael, Joachim Hertzberg, Malik Ghallab, and Martha E. Pollack (editors): *Advances in Plan-Based Control of Robotic Agents*, pages 123–139, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg, ISBN 978-3-540-37724-5. 1

[16] Luckcuck, Matt, Marie Farrell, Louise A Dennis, Clare Dixon, and Michael Fisher: *Formal specification and verification of autonomous robotic systems: A survey*. ACM Computing Surveys (CSUR), 52(5):1–41, 2019. 1, 2, 3, 4, 57

[17] Quigley, Morgan, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, *et al.*: *Ros: an open-source robot operating system*. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009. 2

[18] Paraschos, Alexandros, Nikolaos I Spanoudakis, and Michail G Lagoudakis: *Model-driven behavior specification for robotic teams*. In *AAMAS*, pages 171–178, 2012. 2

[19] Alexandrova, Sonya, Zachary Tatlock, and Maya Cakmak: *Roboflow: A flow-based visual programming language for mobile manipulation tasks*. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5537–5544, 2015. 2

[20] Nordmann, Arne, Nico Hochgeschwender, and Sebastian Wrede: *A survey on domain-specific languages in robotics*. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 8810:195–206, 2014, ISSN 16113349. 2

[21] Hamann, H.: *Swarm Robotics: A Formal Approach.* Springer International Publishing, 2018, ISBN 9783319745282. `https://books.google.com.br/books?id=pnNLDwAAQBAJ`. 2

[22] De Nicola, Rocco, Luca Di Stefano, and Omar Inverso: *Toward formal models and languages for verifiable multi-robot systems.* Frontiers Robotics AI, 5(SEP):1–14, 2018, ISSN 22969144. 3

[23] Bresciani, Paolo, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos: *Tropos: An agent-oriented software development methodology.* Autonomous Agents and Multi-Agent Systems, 8(3):203–236, 2004. 3

[24] Dardenne, Anne, Axel Van Lamsweerde, and Stephen Fickas: *Goal-directed requirements acquisition.* Science of computer programming, 20(1-2):3–50, 1993. 3

[25] Yu, Eric: *Modeling strategic relationships for process reengineering.* Social Modeling for Requirements Engineering, 11(2011):66–87, 2011. 3

[26] Höller, Daniel, Gregor Behnke, Pascal Bercher, Susanne Biundo, Humbert Fiorino, Damien Pellier, and Ron Alford: *Hddl: An extension to pddl for expressing hierarchical planning problems.* Proceedings of the AAAI Conference on Artificial Intelligence, 34(06):9883–9891, Apr. 2020. `https://ojs.aaai.org/index.php/AAAI/article/view/6542`. 3

[27] Abrial, J. R., M. K. O. Lee, D. S. Neilson, P. N. Scharbach, and I. H. Sørensen: *The b-method.* In Prehn, Søren and Hans Toetenel (editors): *VDM '91 Formal Software Development Methods*, pages 398–405, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg, ISBN 978-3-540-46456-3. 3

[28] Mohammed, Ammar, Ulrich Furbach, and Frieder Stolzenburg: *Multi-robot systems: Modeling, specification, and model checking.* Robot Soccer, pages 241–265, 2010. 3

[29] Schillinger, Philipp, Mathias Bürger, and Dimos V Dimarogonas: *Simultaneous task allocation and planning for temporal logic goals in heterogeneous multi-robot systems.* The international journal of robotics research, 37(7):818–838, 2018. 3

[30] Bagade, P., A. Banerjee, and S.K.S. Gupta: *Chapter 12 - validation, verification, and formal methods for cyber-physical systems.* In Song, Houbing, Danda B. Rawat, Sabina Jeschke, and Christian Brecher (editors): *Cyber-Physical Systems*, Intelligent Data-Centric Systems, pages 175–191. Academic Press, Boston, 2017, ISBN 978-0-12-803801-7. `https://www.sciencedirect.com/science/article/pii/B9780128038017000122`. 4

[31] Aniculaesei, Adina, Daniel Arnsberger, Falk Howar, and Andreas Rausch: *Towards the verification of safety-critical autonomous systems in dynamic environments.* In Kargahi, Mehdi and Ashutosh Trivedi (editors): *Proceedings of the The First Workshop on Verification and Validation of Cyber-Physical Systems, V2CPS@IFM 2016, Reykjavík, Iceland, June 4-5, 2016*, volume 232 of *EPTCS*, pages 79–90, 2016. `https://doi.org/10.4204/EPTCS.232.10`. 4, 14

[32] Halder, Raju, José Proença, Nuno Macedo, and André Santos: *Formal verification of ros-based robotic applications using timed-automata*. In *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, pages 44–50, 2017. 4, 14

[33] Ferrari, Alessio, Franco Mazzanti, Davide Basile, Maurice H ter Beek, and Alessandro Fantechi: *Comparing formal tools for system design: a judgment study*. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 62–74, 2020. 5

[34] Konur, Savas, Clare Dixon, and Michael Fisher: *Formal verification of probabilistic swarm behaviours*. In Dorigo, Marco, Mauro Birattari, Gianni A. Di Caro, René Doursat, Andries P. Engelbrecht, Dario Floreano, Luca Maria Gambardella, Roderich Groß, Erol Şahin, Hiroki Sayama, and Thomas Stützle (editors): *Swarm Intelligence*, pages 440–447, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg, ISBN 978-3-642-15461-4. 5, 6

[35] Zhang, Mingzhuo, Dehui Du, Augusto Sampaio, Ana Cavalcanti, Madiel Conserva Filho, and Menghan Zhang: *Transforming robosim models into uppaal*. pages 79–86, 2021. 5, 58, 66

[36] Broy, M.: *Declarative specification and declarative programming*. In *Proceedings of the Sixth International Workshop on Software Specification and Design*, pages 2,3,4,5,6,7,8,9,10,11, Los Alamitos, CA, USA, oct 1991. IEEE Computer Society. https://doi.ieeecomputersociety.org/10.1109/IWSSD.1991.213082. 6

[37] Lestingi, Livia, Cristian Sbrolli, Pasquale Scarmozzino, Giorgio Romeo, Marcello M Bersani, and Matteo Rossi: *Formal modeling and verification of multi-robot interactive scenarios in service settings*. In *Proceedings of the IEEE/ACM 10th International Conference on Formal Methods in Software Engineering*, pages 80–90, 2022. 9

[38] Vogel, Thomas, Marc Carwehl, Genaína Nunes Rodrigues, and Lars Grunske: *A property specification pattern catalog for real-time system verification with uppaal*. Information and Software Technology, 154:107100, 2023. 9

[39] Ravn, Anders P., Jiří Srba, and Saleem Vighio: *Modelling and verification of web services business activity protocol*. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software*, TACAS'11/ETAPS'11, page 357–371, Berlin, Heidelberg, 2011. Springer-Verlag, ISBN 9783642198342. 9

[40] Havelund, Klaus, Arne Skou, Kim Guldstrand Larsen, and Kristian Lund: *Formal modeling and analysis of an audio/video protocol: An industrial case study using uppaal*. In *Proceedings Real-Time Systems Symposium*, pages 2–13. IEEE, 1997. 9

[41] Behrmann, Gerd, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi: *Developing uppaal over 15 years*. Software: Practice and Experience, 41(2):133–142, 2011. 14

[42] Bengtsson, Johan, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi: *Uppaal — a tool suite for automatic verification of real-time systems.* In Alur, Rajeev, Thomas A. Henzinger, and Eduardo D. Sontag (editors): *Hybrid Systems III*, pages 232–243, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg, ISBN 978-3-540-68334-6. 14, 21

[43] Alur, Rajeev and David Dill: *Automata for modeling real-time systems.* In Paterson, Michael S. (editor): *Automata, Languages and Programming*, pages 322–335, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg, ISBN 978-3-540-47159-2. 14, 19

[44] Li, Ran, Jiaqi Yin, and Huibiao Zhu: *Modeling and analysis of rabbitmq using UPPAAL.* In Wang, Guojun, Ryan K. L. Ko, Md. Zakirul Alam Bhuiyan, and Yi Pan (editors): *19th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2020*, pages 79–86, Guangzhou, China, 2020. IEEE. `https://doi.org/10.1109/TrustCom50675.2020.00024`. 15

[45] Askarpour, Mehrnoosh, Christos Tsigkanos, Claudio Menghi, Radu Calinescu, Patrizio Pelliccione, Sergio García, Ricardo Caldas, Tim J von Oertzen, Manuel Wimmer, Luca Berardinelli, Matteo Rossi, Marcello M. Bersani, and Gabriel S. Rodrigues: *Robomax: Robotic mission adaptation exemplars.* In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 245–251, 2021. 40

[46] Koluacik, Deniz: *Koluacik/uppaal-py: Uppaal wrapper for python.* `https://github.com/koluacik/uppaal-py`. 40, 44

[47] Koluacik, Deniz: *Koluacik/uppaal-py: Uppaal wrapper for python - pip library.* `https://pypi.org/project/uppaal-py/0.0.1/`. 40

[48] Bispo, Danilo: *Danilobispo/mutrose-uppaal-generator github project.* `https://github.com/danilobispo/MutRoSe-UPPAAL-Generator`. 40, 44, 55, 56

[49] Bispo, Danilo: *Danilobispo/mutrose-mission-decomposer: A mission decomposer for the mutrose framework. this is built upon the panda hddl parser (https://github.com/panda-planner-dev/pandapiparser).* `https://github.com/danilobispo/MutRoSe-Mission-Decomposer`. 40

[50] (jiffyclub), Matt Davis: *Snakeviz.* `https://jiffyclub.github.io/snakeviz/#snakeviz`. 44

[51] *Python profilers: profile and cprofile module reference.* `https://docs.python.org/3/library/profile.html#module-cProfile`. 44

[52] Holzmann, Gerard J.: *The model checker spin.* IEEE Transactions on software engineering, 23(5):279–295, 1997. 57, 63

[53] Kwiatkowska, Marta, Gethin Norman, and David Parker: *Prism: Probabilistic symbolic model checker.* In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 200–204. Springer, 2002. 57

[54] Cavalcanti, Ana, P Ribeiro, A Miyazawa, A Sampaio, Madiel Conserva Filho, and Andre Didier: *Robosim reference manual.* Technical report, Technical report, University of York, 2019. 58

[55] Dixon, Clare, Alan F. T. Winfield, Michael Fisher, and Chengxiu Zeng: *Towards temporal verification of swarm robotic systems.* Robot. Auton. Syst., 60(11):1429–1441, nov 2012, ISSN 0921-8890. `https://doi.org/10.1016/j.robot.2012.03.003`. 58

[56] Kim, Moonzoo, Kyo Chul Kang, and Hyoungki Lee: *Formal verification of robot movements - a case study on home service robot shr100.* In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pages 4739–4744, 2005. 59, 66, 67

[57] Bouali, Amar: *Xeve , an Esterel verification environment*, pages 500–504. April 2006, ISBN 978-3-540-64608-2. 59

[58] Bensalem, Saddek, Lavindra de Silva, Andreas Griesmayer, François Ingrand, Axel Legay, and Rongjie Yan: *A formal approach for incremental construction with an application to autonomous robotic systems.* Volume 6708, pages 116–132, June 2011. 60, 66, 67

[59] *Rigorous design of component-based systems - the bip component framework*, Apr 2022. `https://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html?lang=en`. 60

[60] Bensalem, Saddek, Andreas Griesmayer, Axel Legay, Thanh Hung Nguyen, Joseph Sifakis, and Rongjie Yan: *D-finder 2: Towards efficient correctness of incremental design.* pages 453–458, April 2011, ISBN 978-3-642-20397-8. 60

[61] MacKenzie, Douglas and Ronald Arkin: *Evaluating the usability of robot programming toolsets.* The International Journal of Robotics Research, 17, September 2001. 61

[62] Levine, John: *Flex & Bison: Text Processing Tools.* " O'Reilly Media, Inc.", 2009. 61

[63] Heinzemann, Christian and Ralph Lange: *vtsl - a formally verifiable dsl for specifying robot tasks.* In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 8308–8314, 2018. 62, 66, 67

[64] Lyons, Damian, Ronald Arkin, S. Jiang, D. Harrington, and T. Liu: *Verifying and validating multirobot missions.* IEEE International Conference on Intelligent Robots and Systems, pages 1495–1502, October 2014. 63

[65] Koubâa, Anis *et al.*: *Robot Operating System (ROS).*, volume 1. Springer, 2017. 63

[66] Pecheur, Charles and Reid Simmons: *From livingstone to smv.* In *International Workshop on Formal Approaches to Agent-Based Systems*, pages 103–113. Springer, 2000. 64, 66, 67

[67] Simmons, Reid and Charles Pecheur: *Automating model checking for autonomous systems.* In *In AAAI Spring Symposium on Real-Time Autonomous Systems*, 2000. 64, 66, 67

[68] Webster, Matthew, David Western, Dejanira Araiza Illan, Clare Dixon, Kerstin I Eder, Michael Fisher, and Anthony G Pipe: *A corroborative approach to verification and validation of human–robot teams*. International Journal of Robotics Research (IJRR), November 2019, ISSN 1741-3176. 65
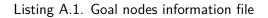
Appendix A

A.1   Files derived from MutRoSe execution

This section includes a more detailed explanation for files derived from MutRoSe execution and how they are used within this work. It is important to note that these files only parse the information contained in the original specification files in order to aid the generation process afterwards.

An excerpt of the goal nodes information file is displayed in Listing A.1. The GM order tree file contains a parent node and its children in each line of the file, separated by an arrow ($-->$). If there is more than one child for the current node, they are separated by white spaces. It is worth noting that some nodes contain a special notation of type $\_OP$ where OP denotes the node runtime operator (e.g. sequential (;), parallel (#), OR (OR) and fallback (FALLBACK), this is used during later stages of generation to create patterns which reflect the changes in the node execution order inside UPPAAL. This file is then parsed as a list of lists where its possible to traverse through nodes similarly as a tree. An example of the goal model order file is displayed in Listing A.2. One can see the similarities when comparing the ordering of goals and tasks (i.e. the mission ordering) between Listings A.2 and A.1 with Figure 3.2, where the latter is the visual representation where the goal order files are derived from.

For the domain definition, the main generated files are: the types and variables information file, a snippet of this file is displayed in Listing A.4. Next, the available methods for an abstract task file is available in Listing A.3. Finally, the method orderings is available in Listing A.5. If the actions contain preconditions or effects, those will be appended to the order along with the type and the predicate or capability. This is done to assure that proper transitions with guards or updates will be generated in UPPAAL during the creation of templates. The file is somewhat illegible for human-reading due to being a direct product for generation of templates. For each of the methods and respective orderings, a corresponding template is automatically generated during the generation stage.

```
Node: G1: Deliver Food to Patients [G2;G3]
```

77

```
Context:
        No Context
Parameters:
Group? 1
Divisible? 1
...
Node: AT3: DeliverToFetch
Context:
        No Context
Parameters:
        Param: current_patient
Group? 1
Divisible? 1
```

Listing A.1. Goal nodes information file

```
1  G1_; --> G2 G3_;
2  G2 -->
3  G3_; --> G4_; G7_;
4  G4_; --> G5 G6
5  G5 -->
6  G6 --> AT1_1
7  AT1_1 -->
8  G7_; --> G8 G9 G10_OR
9  G8 -->
10 G9 -->
11 G10_OR --> G11 G12
12 G11 --> AT2_1
13 AT2_1 -->
14 G12 --> AT3_1
15 AT3_1 -->
```

Listing A.2. Goal model order

```
Name: GetFood
food-pickup
Name: DeliverToTable
table-deliver
Name: DeliverToFetch
```

```
fetch - deliver
Name: FetchMeal
fetch - meal - with - human
fetch - meal - with - robot
...
```

Listing A.3. Excerpt from methods avaliable for Abstact Tasks (AT) in food logistics mission

```
Variable name: ?r  Variable  Type:  robot
Variable name: ?l  Variable  Type:  location
Variable name: ?d  Variable  Type:  delivery
Variable name: ?r  Variable  Type:  robot
...
```

Listing A.4. Excerpt from types and variables information file

```
Method name: food - pickup
navto wait - for - food  ___task_effect pickedmeal_true_argument__?r
Method name: table - deliver
___method_precondition_table - deliver pickedmeal_true_argument__?r navto approach - patient - table deliver - to
    - table  ___method_capability _argument_manipulation
Method name: fetch - deliver
___method_precondition_fetch - deliver pickedmeal_true_argument__?r1 navto FetchMeal
Method name: fetch - meal - with - human
___method_precondition_fetch - meal - with - human patientcanfetch_true_argument__?p approach - human wait - for -
    human - to - fetch  ___task_effect pickedmeal_false_argument__?r
Method name: fetch - meal - with - robot
navto approach - robot grasp - meal ___task_effect pickedmeal_false_argument__?r2 ___task_effect
    pickedmeal_true_argument__?r1 deliver - meal - to - patient ___task_effect pickedmeal_false_argument__?r
...
```

Listing A.5. Snippet from method orderings file

A.2   Domain files

This section includes the domain files used for the missions included in the Chapter 4.
As such, they do not require further explanation in the appendix as they are already
explained in the results.

```
1  ( define ( domain hospital )
2      (: types
3              delivery pickup patient location - object
4      )
5      (: predicates
6              ( patientcanfetch ?p - patient )
7              ( patientcanopen ?p - patient )
8              ( deliverypatient ?p - patient ?d - delivery )
9              ( deliverylocation ?l - location ?d - delivery )
10             ( pickuppatient ?p - patient ?pk - pickup )
```

```
11              (pickuplocation ?l - location ?pk - pickup)
12              (pickeddishes ?r - robot)
13              (pickedmeal ?r - robot)
14              (at ?r - robot ?l - location)
15         )
16         (: capabilities manipulation door-opening)
17
18         (: task GetFood : parameters (?r - robot ?l - location ?d - delivery))
19         (: method food-pickup
20              : parameters (?r - robot ?l - location ?d -delivery)
21              : task (GetFood ?r ?l ?d)
22              : ordered-subtasks (and
23                   (navto ?r ?l)
24                   (wait-for-food ?r ?l ?d)
25              )
26         )
27         (: action wait-for-food
28              : parameters (?r - robot ?l - location ?d - delivery)
29              : effect (and
30                   (pickedmeal ?r)
31              )
32         )
33
34         (: task DeliverToTable : parameters (?r - robot ?l - location ?p - patient))
35         (: method table-deliver
36              : parameters (?r - robot ?l - location ?p - patient)
37              : task (DeliverToTable ?r ?l ?p)
38              : precondition (and
39                   (pickedmeal ?r)
40              )
41              : ordered-subtasks (and
42                   (navto ?r ?l)
43                   (approach-patient-table ?r ?l ?p)
44                   (deliver-to-table ?r ?l)
45              )
46         )
47         (: action approach-patient-table
48              : parameters (?r - robot ?l - location ?p - patient)
49         )
50         (: action deliver-to-table
51              : parameters (?r - robot ?l - location)
52              : required-capabilities (manipulation)
53         )
54
55         (: task DeliverToFetch : parameters (?r1 ?r2 - robot ?l - location ?p -
           patient))
56         (: method fetch-deliver
57              : parameters (?r1 ?r2 - robot ?l - location ?p - patient)
58              : task (DeliverToFetch ?r1 ?r2 ?l ?p)
```

```
59                      : precondition (and
60                              (pickedmeal ?r1)
61                      )
62                      : ordered - subtasks (and
63                              (navto ?r1 ?l)
64                              (FetchMeal ?r1 ?r2 ?l ?p)
65                      )
66              )
67
68      (:task FetchMeal :parameters (?r1 ?r2 - robot ?l - location ?p - patient))
69      (:method fetch -meal -with -human
70              :parameters (?r1 ?r2 - robot ?l - location ?p - patient)
71              :task (FetchMeal ?r1 ?r2 ?l ?p)
72              :precondition (and
73                      (patientcanfetch ?p)
74              )
75              :ordered - subtasks (and
76                      (approach -human ?r1 ?l ?p)
77                      (wait - for -human -to - fetch ?r1 ?l ?p)
78              )
79      )
80      (:method fetch -meal -with -robot
81              :parameters (?r1 ?r2 - robot ?l - location ?p - patient)
82              :task (FetchMeal ?r1 ?r2 ?l ?p)
83              :ordered - subtasks (and
84                      (navto ?r2 ?l)
85                      (approach -robot ?r1 ?r2)
86                      (grasp -meal ?r2 ?r1)
87                      (deliver -meal -to - patient ?r2 ?p ?l)
88              )
89      )
90
91      (:task PickupDishes :parameters (?r1 ?r2 - robot ?l - location ?p -
        patient))
92      (:method pickup -with -door -opening
93              :parameters (?r1 ?r2 - robot ?l - location ?p - patient)
94              :task (PickupDishes ?r1 ?r2 ?l ?p)
95              :precondition (and
96                      (not (pickeddishes ?r1))
97              )
98              :ordered - subtasks (and
99                      (navto ?r1 ?l)
100                     (navto ?r2 ?l)
101                     (approach -door ?r1 ?l)
102                     (approach -door ?r2 ?l)
103                     (open -door ?r1 ?r2 ?l)
104                     (PickDishesTwoRobotsAtLocation ?r1 ?r2 ?l ?p)
105             )
106     )
```

```
107         (: method pickup - without - door - opening
108                 : parameters (? r1 ?r2 - robot ?l - location ?p - patient )
109                 : task ( PickupDishes ?r1 ?r2 ?l ?p )
110                 : precondition ( and
111                         ( patientcanopen ?p )
112                         ( not ( pickeddishes ?r1 ) )
113                 )
114                 : ordered - subtasks ( and
115                         ( navto ?r1 ?l )
116                         ( approach - door ?r1 )
117                         ( wait - for - door - opening ?r1 )
118                         ( PickDishesOneRobotAtLocation ?r1 ?r2 ?l ?p )
119                 )
120         )
121
122         (: task PickDishesTwoRobotsAtLocation : parameters (? r1 ?r2 - robot ?l -
            location ?p - patient ))
123         (: method pick - dishes - two - robots - at - location
124                 : parameters (? r1 ?r2 - robot ?l - location ?p - patient )
125                 : task ( PickDishesTwoRobotsAtLocation ?r1 ?r2 ?l ?p )
126                 : precondition ( and
127                         ( at ?r1 ?l )
128                         ( at ?r2 ?l )
129                 )
130                 : ordered - subtasks ( and
131                         ( PickDishes ?r1 ?r2 ?l ?p )
132                 )
133         )
134
135         (: task PickDishesOneRobotAtLocation : parameters (? r1 ?r2 - robot ?l -
            location ?p - patient ))
136         (: method pick - dishes - one - robot - at - location
137                 : parameters (? r1 ?r2 - robot ?l - location ?p - patient )
138                 : task ( PickDishesOneRobotAtLocation ?r1 ?r2 ?l ?p )
139                 : precondition ( and
140                         ( at ?r1 ?l )
141                         ( not ( at ?r2 ?l ) )
142                 )
143                 : ordered - subtasks ( and
144                         ( PickDishes ?r1 ?r2 ?l ?p )
145                 )
146         )
147
148         (: task PickDishes : parameters (? r1 ?r2 - robot ?l - location ?p -
            patient ))
149         (: method pick - dishes - with - human
150                 : parameters (? r1 ?r2 - robot ?l - location ?p - patient )
151                 : task ( PickDishes ?r1 ?r2 ?l ?p )
152                 : ordered - subtasks ( and
```

```
153                            (approach-human ?r1 ?l ?p)
154                            (wait-for-human-to-place-dish ?r1 ?p)
155                    )
156            )
157        (:method pick-dishes-with-robot-at-location
158                    :parameters (?r1 ?r2 - robot ?l - location ?p - patient)
159                    :task (PickDishes ?r1 ?r2 ?l ?p)
160                    :precondition (and
161                            (at ?r2 ?l)
162                    )
163                    :ordered-subtasks (and
164                            (pick-patient-dishes ?r2 ?p)
165                            (load-dishes ?r2 ?r1)
166                    )
167        )
168        (:method pick-dishes-with-robot-not-at-location
169                    :parameters (?r1 ?r2 - robot ?l - location ?p - patient)
170                    :task (PickDishes ?r1 ?r2 ?l ?p)
171                    :precondition (and
172                            (not (at ?r2 ?l))
173                    )
174                    :ordered-subtasks (and
175                            (navto ?r2 ?l)
176                            (pick-patient-dishes ?r2 ?p)
177                            (load-dishes ?r2 ?r1)
178                    )
179        )
180
181        (:task RetrieveDishes :parameters (?r - robot ?l - location))
182        (:method dishes-retrieval
183                    :parameters (?r - robot ?l - location)
184                    :task (RetrieveDishes ?r ?l)
185                    :ordered-subtasks (and
186                            (navto ?r ?l)
187                            (retrieve-dishes ?r ?l)
188                    )
189        )
190
191        (:action approach-human
192                    :parameters (?r - robot ?l - location ?p - patient)
193        )
194        (:action approach-robot
195                    :parameters (?r1 ?r2 - robot)
196        )
197        (:action grasp-meal
198                    :parameters (?r1 ?r2 - robot)
199                    :effect (and
200                            (not (pickedmeal ?r2))
201                            (pickedmeal ?r1)
```

```
202                    )
203            )
204        (: action  deliver - meal - to - patient
205                : parameters  (? r  -  robot  ?p  -  patient  ?l  -  location )
206                : effect  (and
207                        (not  ( pickedmeal  ?r ))
208                )
209        )
210        (: action  wait - for - human- to - fetch
211                : parameters  (? r  -  robot  ?l  -  location  ?p  -  patient )
212                : effect  (and
213                        (not  ( pickedmeal  ?r ))
214                )
215        )
216
217        (: action  wait - for - human- to - place - dish
218                : parameters  (? r  -  robot  ?p  -  patient )
219                : effect  (and
220                        ( pickeddishes  ?r )
221                )
222        )
223        (: action  pick - patient - dishes
224                : parameters  (? r  -  robot  ?p  -  patient )
225                : effect  (and
226                        ( pickeddishes  ?r )
227                )
228        )
229        (: action  load - dishes
230                : parameters  (? r1  ?r2  -  robot )
231                : effect  (and
232                        (not  ( pickeddishes  ?r1 ))
233                        ( pickeddishes  ?r2 )
234                )
235        )
236        (: action  approach - door
237                : parameters  (? r1  -  robot  ?l  -  location )
238        )
239        (: action  open - door
240                : parameters  (? r1  ?r2  -  robot  ?l  -  location )
241                : required - capabilities  (door - opening )
242        )
243        (: action  wait - for - door - opening
244                : parameters  (? r  -  robot )
245        )
246        (: action  pickup - dishes - with - robot
247                : parameters  (? r1  ?r2  -  robot  ?l  -  location )
248                : effect  (and
249                        ( pickeddishes  ?r1 )
250                )
```

```
251          )
252          (: action retrieve - dishes
253                   : parameters (? r - robot ?l - location)
254          )
255
256          (: action navto
257                   : parameters (? r - robot ?l - location)
258          )
259 )
```

Listing A.6. Domain definition file for food logistics example in HDDL

```
1  (define (domain hospital)
2          (: types location storage obj agent - object)
3          (: predicates
4                   (requestingequipment ?a - agent)
5                   (at ?o - obj ?s - storage)
6                   (at ?l - location ?a - agent)
7                   (requested ?o - obj ?a - agent)
8          )
9          (: capabilities )
10
11         (: task GetObject : parameters (? r - robot ?s - storage ?o - obj))
12         (: method object - get
13                  : parameters (? r - robot ?s - storage ?o - obj)
14                  : task (GetObject ?r ?s ?o)
15                  : precondition ()
16                  : ordered - subtasks (and
17                          (get - object ?r ?s ?o)
18                  )
19         )
20         (: action get - object
21                  : parameters (? r - robot ?s - storage ?o - obj)
22         )
23
24         (: task RechargeBattery : parameters (? r - robot))
25         (: method battery - recharge
26                  : parameters (? r - robot)
27                  : task (RechargeBattery ?r)
28                  : precondition ()
29                  : ordered - subtasks (and
30                          (recharge - battery ?r)
31                  )
32         )
33         (: action recharge - battery
34                  : parameters (? r - robot)
35         )
36
37         (: task DeliverObjects : parameters (? r - robot ?l - location))
```

85

```
38          (: method objects - delivery
39                  : parameters (?r - robot ?l - location)
40                  : task (DeliverObjects ?r ?l)
41                  : precondition ()
42                  : ordered - subtasks (and
43                          (deliver - objects ?r ?l)
44                  )
45          )
46          (: action deliver - objects
47                  : parameters (?r - robot ?l - location)
48          )
49
50          (: task ReturnObjectsToCheckpoint : parameters (?r - robot))
51          (: method object - returning
52                  : parameters (?r - robot)
53                  : task (ReturnObjectsToCheckpoint ?r)
54                  : precondition ()
55                  : ordered - subtasks (and
56                          (return - objects ?r)
57                  )
58          )
59          (: action return - objects
60                  : parameters (?r - robot)
61          )
62
63          (: task AlertTrigger : parameters (?r - robot))
64          (: method alert - trigger
65                  : parameters (?r - robot)
66                  : task (AlertTrigger ?r)
67                  : precondition ()
68                  : ordered - subtasks (and
69                          (trigger - alert ?r)
70                  )
71          )
72          (: action trigger - alert
73                  : parameters (?r - robot)
74          )
75 )
```

Listing A.7. Domain definition file for deliver goods - equipment mission in HDDL